TRANSPORT AND OVERLAY PROTOCOLS FOR HIGH-BANDWIDTH APPLICATIONS OVER THE INTERNET.

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Vidhyashankar Venkataraman

August 2009

TRANSPORT AND OVERLAY PROTOCOLS FOR HIGH-BANDWIDTH APPLICATIONS OVER THE INTERNET.

Vidhyashankar Venkataraman, Ph.D.

Cornell University 2009

The Internet has truly been one of the most dominant discoveries over the past two decades. One of the key reasons to its massive success has been the evolution of communication technologies: for instance, long-haul optical links now connect networks across the world and enable communication at high speeds. The increase in network capacities have also given rise to a wide range of high-bandwidth applications such as file sharing and media streaming. For these applications to sustain at such high capacities, the underlying communication protocols should also be utilizing as much of the bandwidth as possible. In this thesis, we propose two end-to-end communication protocols that attain this goal. One is a unicast (one-to-one) transport protocol while the other is an overlay multicast (one-to-many) protocol.

Long haul networks suffer from the well-known limitations of TCP over long fat pipes. High-performance protocols like XCP require changes in the network. Other protocols like FastTCP assume nothing about the network but may not perform as well as network-aware protocols. In the first part of my thesis, we present Priority Layered Transport (PLT), a transport protocol that exploits the fact that networks can offer priority queuing, thus finding the sweet spot between assuming too much and too little about the network. Our protocol splits a given transport flow into two prioritized flows. The higher priority flow operates with the legacy congestion control while the lower priority flow ag-

gressively exploits spare capacity in the network while not interfering with the other participating flows. We show through experiments that our protocol can produce near-perfect goodputs even in lossy networks and changing network conditions.

The rising popularity of IPTV has triggered renewed interest in overlay multicast. Traditional tree-based solutions are known to be complex, fragile and non-robust to churn and heterogeneity in the end hosts' bandwidth capacities. We present Chunkyspread, an unstructured multicast protocol that uses multiple trees to provide fine-grained control over member load, reacts quickly to membership changes, scales well, and has low overhead. This thesis gives a detailed description of Chunkyspread and an apples-to-apples comparison with the state-of-the-art Splitstream multi-tree algorithm. We show that Chunkyspread exhibits far better control over transmit load than Splitstream, while showing comparable or better latency and responsiveness to churn. This comparison establishes Chunkyspread as the 'best of breed' among tree-based P2P multicast algorithms.

BIOGRAPHICAL SKETCH

Vidhyashankar Venkataraman (aka Vidhya) was born and brought up in Chennai, India. Driven by his passion for math and science, he joined the Indian Institute of Technology, Madras in 2000 and pursued a bachelor's degree in Computer Science. Pushed by his peers, his professors and more importantly his increased interest in Computer Science, he applied for graduate studies at Cornell. He started his PhD in 2003 and has been found loitering in Upson Hall ever since. He is now heading to Yahoo for a career in the field that he chose ten years ago.

To appa, amma and Harish.

ACKNOWLEDGEMENTS

The years that I have spent in Cornell have been some of my most important ones in my life. There are quite a few people that I am indebted to for reaching the finish line. Paul Francis, needless to say, is an excellent researcher and I have always been awe-struck at the way he can transform simple ideas into brilliant solutions. His direct, frank remarks have helped me propel forward in my research and in my life. Dr. T. V. Lakshman was the co-advisor for my work on PLT. He is one of the most humble and affable researchers I have met and the most influential in my life. I spent my summers of 2005 and 2006 working in Lucent Technologies with Lakshman and Murali Kodialam. I was low on confidence and morale before I started work at Lucent and they were big sources of inspiration in my determination to move forward.

Of course, this work would not have been possible without support from peers. I would like to specifically thank Kaouru Yoshida for his help in Splitstream simulations and George Kurian for his inputs in building the PLT codebase. Vivek for attending my practice talks, proof-reading my papers and helping me when I had doubts on Swaplinks; he has been of great help almost throughout my PhD. Then there are the three important people I want to acknowledge from the bottom of my heart: one is Mahesh who has been a great friend of mine both personally and professionally. Then there are Karthick and Chandra, my apartment mates who had been an integral part of my graduate school life and had often provided me great company when the chips were down. I also thank Ashwin for patiently sitting through my practice talks and providing some wonderful feedback each time. And Manpreet for helping me bootstart with PLT.

I am also glad I made some great friends inside the department and in Bell Labs. Special mentions should go to Lucian, Art, Panda, Hitesh, Maya, Moon, Benyah, Sumit and Anand. I have also had a great group of friends to hang out with outside of Upson. I can't forget the Friday night poker and card games with Ishani, Niranjan, and Bhragavi. Giri and Lakshmi had provided great company during my initial years: I will still remember the time when they drove from Albany to spend the new year in Ithaca with me when I was working for a deadline all alone. Rama was also a great friend during my initial years. I should also thank the six girls from Highland for letting me freeride on their food during my final year at Cornell when most of my friends had graduated.

Over and above all the aforementioned people, I would like to thank my amma, appa and Harish for all the love and affection they have showered me throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The Internet is a vast packet switched network that enables communication across remote corners of the globe. From an experimental test bed in the late seventies to a ubiquitous entity in recent years, the Internet has truly come a long way. This has been made possible due to contributions from both the academia and the industry.

The original architects of the Internet divided it into five distinct layers and encapsulated a set of functions into each layer for the sake of interoperability across the layers. The model consisted of the application, transport, network, data-link and the physical layers. Each layer provided some service to the layer above it and received service from the layer below it.

Applications from two end hosts communicate with each other by passing data down the layers at the sender and finally back up at the receiver (Figure 1.1). This Internet model is commonly called the hourglass model; the upper part represents different kinds of applications, the lower part represents various types of physical communication technologies while the narrow waist is the Internet protocol (IP) and is the common protocol designed to provide interoperability between the upper and the lower parts (Figure 1.2).

This layered hourglass model applies to the Internet even today. One difference now is that the two ends of the hourglass, namely the application and the physical layers, have expanded considerably. The application layer is richer and more complex with respect to its functionalities and requirements. For example, a live streaming video application needs low latencies, steady arrival of the

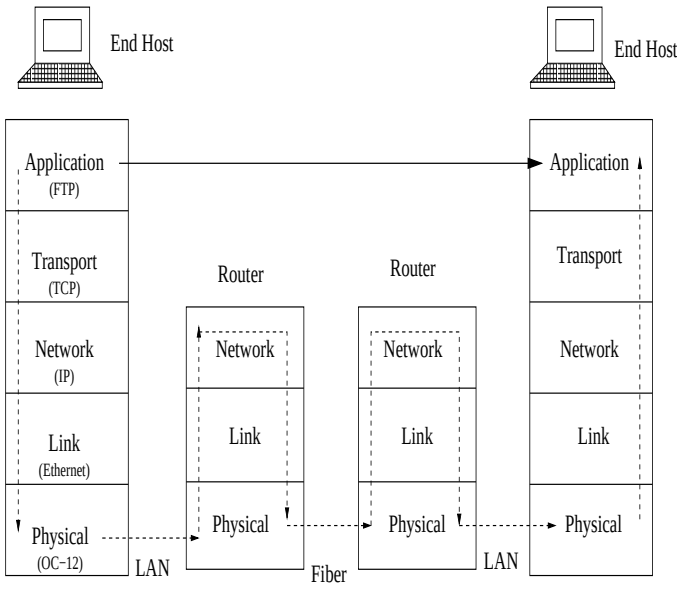


Figure 1.1: Communication between two end hosts using the layered model. The dotted line indicates the flight of the packet. The solid line indicates the communication that was originally intended. Instance of the layers are provided within parentheses.

video frames (low jitter), sufficient bandwidth capacities at the client and moderate reliability guarantees. An update system associated with a stock exchange may need low bandwidth but requires real-time and strict reliability guarantees. An online game involving multiple players across the Internet should be able to provide reliable and real-time updates. As a sidenote, the top of the hourglass has also become taller with the advent of virtual private networks, overlays and other tunneling technologies. The incorporation of functionalities such as security and address translations has only made it more complex.

At the other end of the hourglass, the physical and link layers have also evolved and diversified. The Internet started out with a 32Kbps-backbone ([1]). Over the years, network capacities have seen an exponential growth. Now, even DSL modems and broadband connections can offer bandwidth capacities of be-

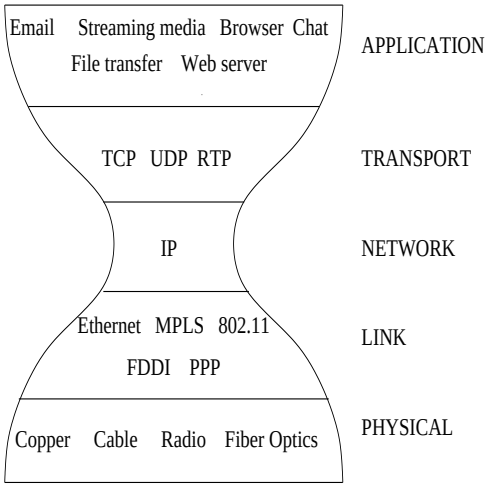


Figure 1.2: The Internet Hourglass: Sometimes there could be one or more layers between the transport and the application layers. For example, a peer-to-peer substrate, an overlay multicast protocol are middlewares that mask the transport from the applications.

tween 250Kbps and 24Mbps ([1]) to users while a user in a corporate or University LAN can enjoy bandwidth capacities of upto 10 Gbps. Internet2 ([2]) is a research test bed that operates on a 100 Gbps backbone. 3G (mobile) networks can provide capacities in the lower tens of Mbps.

With both ends of the hourglass witnessing such an immense growth, it is important that the waist and the layers near the waist, namely the TCP/IP (and the middleware) layers, be efficient in utilizing the underlying bandwidth so as not to limit the evolution of the application layer. This would ensure that the evolution of the application layer is only limited by the physical layer and not by the layers in between.

In fact, a key contributor to the increase in complexity of the application layer has been the steady improvement in bandwidth capacities of the underlying network. Applications such as streaming media and file sharing need good

bandwidth capacities at the client's connection to the Internet (also called the access link). On the infrastructure side, corporate web servers, data centers situated across the globe, and content delivery networks such as Akamai need 'fat' network pipes to sustain themselves. These applications undervalue the need for utilizing the network efficiently. In recent times, advances in cloud computing have led to a change in the perception of a client machine: the Internet cloud is the new computer which runs the user applications in the form of web services while the client runs a lightweight operating system. Such a paradigm shift should be transparent to the user. For this to be possible, any unnecessary bottlenecks in the transport, network and the middleware layers should be avoided.

1.1 Fixing the waist

The growth of the Internet revealed some architectural limitations and posed challenges to the research community. Numerous works have addressed Internet's problems on high speed congestion control [30], mobility [10], quality-of-service [11] and its evolvability [12]. These *in-the-network* solutions require architectural changes and hence find few takers: Internet Service Providers (ISPs) have little motivation or incentive to deploy a new architecture and risk disturbing a working system. Taking this limitation as a given, we were seeking *end-to-end* approaches (operating at layers above the network layer) to enable bandwidth-hungry applications make maximal use of the increasing bandwidth capacities.

In this thesis, we present two end-to-end protocols that achieve this goal. One is a solution at the transport layer called the Priority Layered Transport

(PLT). TCP is known to not work well with high-speed networks. TCP's problems have been tackled by many protocols in the past but either not completely or with some caveats. We propose a new protocol that does not have the limitations of TCP and other existing transport protocols.

The other solution is an overlay protocol that operates between the application and the transport layers called Chunkyspread. We explore the challenges thrown in when hosts share the burden of sending data in a distributed fashion. This protocol is an end-to-end solution to multicast (one-to-many) while PLT is an end-to-end solution to unicast (one-to-one).

1.2 PLT

TCP is known to not work well with high speed and satellite networks. This is because of the limitations in TCP's *congestion control* algorithm. Network congestion happens when there are more bytes to be sent than can be handled by the network. And a congestion collapse is said to occur when all participating connections (also called flows) pump more data into the network than what it can handle and end up getting diminishing returns. TCP's congestion control helps to avoid the collapse by making each flow conservatively increase its sending rate to detect additional capacity in the network and back off when congestion is perceived by the protocol. TCP perceives congestion through packet losses. This algorithm is called the Additive Increase Multiplicative Decrease denoted AIMD. Traditionally, TCP's congestion control has done well in maximizing throughput while simultaneously giving equal ($fair$) throughputs amongst all the TCP flows in the network. But in high speed networks, this al-

gorithm is too conservative to quickly ramp up its sending rate. Further, it has been shown that even network losses of as low as one packet loss in every 10^6 transmitted packets ([32]) can be detrimental to TCP's throughput.

Many protocols have been proposed during this decade that alter TCP's AIMD curve to increase its sending rate faster (be more *aggressive*) yet remain *fair* to legacy TCP flows in the network[1]. The conflict between these goals has always remained a point of discussion and numerous works have tried to find that sweet spot where an ideal protocol should operate. This has been difficult to achieve with the purely end-to-end transport protocols. Some works in the past have suggested changes to the underlying network architecture to achieve this purpose: as we had discussed before, these protocols are difficult to deploy. In this thesis, we present a protocol called Priority Layered Transport (PLT) that uses an existing network feature to attain the goal. At the same time, PLT achieves all the other requirements that a congestion control algorithm is expected to satisfy: good throughputs even in lossy environments, good receptiveness to changing network conditions and fairness to fellow TCP and PLT flows.

1.3 Chunkyspread

Multicast protocols offer a distinct advantage over the simple client-server unicast by reducing the burden of the server. Traditional network (IP) multicast has faced criticism for its limited ability to support a large number of nodes and groups (scalability) and deployability.

[1]Qualitatively speaking, a non-TCP flow A (say) is said to be fair to a TCP flow B if B gives at least the same throughput as the case when A was a TCP flow.

The interest in overlay networks led to a plethora of overlay multicast solutions during this decade. These solutions were much more scalable and deployable than IP multicast but wasted more bandwidth and incurred greater latencies. Apart from a few other applications, wide-area multicast solutions primarily targeted live streaming media; the challenges in these solutions were to provide a continued stream to the user in spite of a large number of clients (inter-changeably referred to as nodes or peers) in the network, varied availability of bandwidth capacities amongst the peers (bandwidth heterogeneity), and clients entering and leaving the overlay (churn). The initial solutions solved this problem by creating multicast distribution trees spanning the paticipating hosts. This partly tackled the scalability issue but did not solve the equally, if not more, important challenges of bandwidth heterogeneity and churn in the system. Bandwidth heterogeneity is important because participating peers may use different kinds of physical technologies – they may be behind a broadband connection, DSL modem or a University LAN – whose uplink and downlink bandwidth capacities differ by orders of magnitude. Churn is important since participating peers are known to join and leave in large numbers and the system has to be robust to such scenarios. More recent works have tried to resolve these issues with a limited degree of success. These protocols are described in the next chapter. In this thesis, we present Chunkyspread, an overlay multicast protocol that achieves these goals and does that better than contemporary works.

The thesis is structured as follows. In the next chapter, we give a chronological survey of related works that helped shape PLT and Chunkyspread. We present the two protocols in the following two chapters (Chapters 3 and 4). The thesis concludes with some remarks on future work.

CHAPTER 2

RELATED WORK

Before we describe our work in greater detail, we would like to present a brief chronological survey of important research works in reliable transport, overlay multicast protocols and other related areas.

2.1 Transport protocols

TCP/IP is the communication protocol used on the Internet today. It was adopted as the networking standard for ARPANET in the early eighties only to finally emerge as the dominant protocol suite for the worldwide web in the early nineties and has remained that way. Instances of congestion collapse were observed in the eighties that led to the proposal of congestion control ([16]). At about the same time, TCP Reno ([17]), NewReno ([18]) and SACK ([19]) were proposed to tweak the retransmission mechanism of TCP and improve its throughput ([20]). By convention, TCP always refers to the default TCP-Reno standard and more recently, TCP-SACK.

It should be remembered that at the time of the inception of TCP, the backbone (NSFnet) had capacities of 32-40Kbps which increased to around 45Mbps by 1991. At such bandwidth capacities and even with inter-atlantic delays, TCP's congestion control is usually well-behaved. But as bandwidth capacities of the underlying network increased by orders of magnitude over the next few years, cracks in the algorithm began to show.

During the late nineties, TCP's limitations were exposed in [32] and [33] and

the authors had analytically proven that when the *bandwidth-delay product* of the network is high, it is difficult for TCP to maintain large average congestion windows even at low loss probabilities. The bandwidth-delay product refers to the amount of data that can be sent within a time span of one round trip time; the value is also the total window size that participating connections (also called flows) should maintain in order to guarantee maximum throughput. This value is of the order of 10^4KB in cross-country links in the United States. Examples of networks with a high value are transatlantic optical networks and satellite networks. Such networks are also commonly called *long, fat* networks.

Lakshman and Padhye's observations resulted in a host of alternative proposals to improve the performance of transport protocols in high bandwidth-delay product networks, such as FastTCP (also called FAST) [31], XCP [30], RCP [8], BIC TCP [42], CUBIC [41], HSTCP [35], STCP [36]. These protocols (including TCP) can be broadly classified into two schools: one that needs explicit support from the network and the other one that doesn't need network support but rely on end-to-end signals.

2.1.1 Implicit transport protocols

We first provide a brief description of protocols that rely on end-to-end (implicit) signals. Many protocols use packet loss as a sign for congestion. One limitation with such protocols is that any packet loss is always seen as one due to congestion when losses can occur because of other factors such as channel errors and buffer overflows at the end host or in middle boxes. TCP is an end-to-end protocol that uses packet losses to detect congestion in the network. We

had pointed out earlier that TCP's conservative window growth does not help with networks that require flows to maintain large window sizes.

HSTCP [35] and STCP [36] augment the performance of TCP by altering its congestion algorithm to increase aggressively and decrease gracefully. HSTCP [35] is a modified congestion control of TCP wherein the increase and the decrease parameters are functions of the congestion window: the increase is larger and the decrease smaller when the window size is greater than some threshold. STCP [36] increases multiplicatively if the congestion window size is larger than a threshold and decreases by a factor less than what TCP does. [31] and [37] show that these protocols are unstable in the presence of high network dynamics.

CUBIC TCP [41] is the default standard in recent Linux kernels. It is a derivative of BIC TCP [42] which uses a different kind of congestion growth curve based on a search algorithm for the optimal window size. BIC TCP grows its window using a binary search between the current window size and the window size at which the previous loss event occured. However, a packet loss event still leads to a multiplicative decrease. CUBIC uses a more stable cubic curve for the window growth function, with the inflection point being the equilibrium window size prior to the last congestion event.

Thus far, we have talked of implicit protocols where the end-to-end congestion signal is packet loss. Increase in round trip time is another likely candidate for a congestion signal. Delay-based feedback is believed to offer much more control and stability to the window evolution than the oscillation-prone loss-based feedback. A delay-based congestion control is more error-prone than loss-based congestion control because of the degree of inaccuracy involved in

calculating the round trip time. One interesting point about delay-based congestion control schemes is that they still cut back their windows on encountering packet losses much like their loss-based counterparts.

FastTCP ([31]) revived the interest in delay-based congestion protocols that had initially started with TCP-Vegas ([34]). While TCP-Vegas follows a slower window growth much on the lines of legacy TCP, Fast TCP adopts a much faster growth. In FastTCP, the congestion window value for each flow is maintained such that it is just enough for the bottleneck link to get saturated and start queuing up. Though FastTCP cuts back aggressively during losses which are bound to happen when the network is subject to heavy dynamics, it ramps up multiplicatively, with the increase factor depending on the current congestion window size and the queuing delay; the factor reduces once queueing delay becomes non-zero. [10] shows that such a setup can lead to very fast convergence. FastTCP however relies on the value of the target queuing delay for its good performance. The optimal value is scenario-dependent and a poor value of the delay parameter may cause oscillations in the system.

Compound TCP ([21]) is another delay-based protocol for high bandwidth-delay product networks. Its congestion window size is the sum of two window sizes: an AIMD-based value and a delay-based value. The delay-based window grows much on the lines of FastTCP's window. When there is a perceived increase in the delay, the window is decreased so as to keep the sum of the two window sizes approximately constant. This decrease in window size improves its fairness with TCP-Reno but could result in a loss of aggressiveness under lossy conditions.

Download accelerators are commonly used by internet users to improve

their download speeds. They work on the simple premise that using a number of TCP flows at the same time mask the limitations of TCP. This is quite similar to having a modified AIMD protocol with a constant integer (greater than unity) added every round trip time during the additive phase. The central limitation to most of these products is that their performance depend on the number of flows originated. Further, such a system would adversely affect fairness in the network.

2.1.2 Explicit transport protocols

This form of explicit congestion feedback is in direct contrast to the binary loss-based congestion signal used in TCP. XCP ([30]) is one such protocol. An XCP-enabled router periodically calculates the spare capacity that should be utilized by the flows through it. The router then fairly distributes this capacity across all flows based on their respective congestion window sizes and RTT values and suggests the window change for each flow, as and when packets from the flows traverse through that router. The suggested value is overwritten on the packet only if the value from previous routers in the path is greater than the former. This ensures that the XCP sender eventually receives the window change from the bottleneck link and makes the necessary changes to its window size. Though XCP can be considered as a baseline case for transport protocols in this problem domain, it needs routers to be XCP-compatible for them to work. Further, for XCP to be friendly to other TCP flows, dynamic weighted fair share queuing should be enabled.

RCP ([8]) is another explicit congestion protocol that improves over XCP by

reducing the complexity in an XCP router. RCP is specifically designed for mid-size and short flows on the Internet. End hosts detect the fair-share rate of the bottleneck link by specifying a target rate on each packet. Routers update this value if their rate is less than the indicated value. This requires less modification to routers than XCP.

2.1.3 Transport protocols and packet losses

Transport protocols can suffer from temporary losses in throughput under high lossy network conditions. Non-congestion losses (such as those due to channel errors and router misconfigurations [9], [14], [4]) are common in long-haul optical links where loss rates could go as high as 0.01% ([9]) to 1% ([14]). With an increasing volume of real-time applications like VoIP and IPTV ([15]), cross traffic is also increasingly likely to affect the performance of transport protocols.

2.1.4 Prioritized packet transmission

There has been work in the past that makes use of priority queuing in routers to enhance TCP's performance. TCP-Peach ([38]) is one such protocol designed for satellite networks. TCP-Peach uses dummy segments at the low priority to probe the availability of bandwidth in bottleneck links. The sender gathers feedback from the receiver for these probes and increases its congestion window accordingly. PLT differs from TCP-Peach in actually sending data packets at the low priority so as to utilize the spare bandwidth at the bottleneck quickly.

The general idea of splitting packets of a given flow over multiple priority

levels enjoys a long and varied history. This can especially be seen in layered media encodings, where higher-fidelity, higher-volume information is sent at lower priorities. This allows media quality to degrade gracefully and fairly in the face of congestion. This body of work, however does not apply to reliable transport per se. There have been a few proposals to exploit priority queuing for reliable transport in certain limited contexts. To our knowledge, however, the idea of exploiting both low and high priority channels broadly across the lifetime of a connection has not been proposed.

HSTCP-LP is a congestion control algorithm for bulk data background transfers in fat pipes sent at the low priority. It uses an aggressive delay-based congestion control scheme to ramp up quickly and at the same time maintain fairness with other low priority flows. Fast Start ([39]) modifies TCP's slow start to ensure faster completion of short and bursty data transfers. When a TCP connection is established, the initial values of TCPs parameters such as cwnd, ssthresh are set to the values based on the end hosts cached history. If the initial value of cwnd is greater than unity, TCP enters the fast start phase wherein one packet is sent at a high priority while the remaining number (cwnd-1) of packets are sent through low priority. Fast Start lasts at most for a single RTT during the slow start phase and can stop prematurely if there are multiple losses. In contrast, the low priority traffic of PLT takes part during the entire course of a TCP connection so as to prevent the bottleneck from being under-utilized.

TCP-Nice is a transport protocol specially tuned for large background transfers for which users are not waiting for, such as prefetching and data backup. The congestion algorithm senses the spare bandwidth at the bottleneck and backs off just when foreground transfers start filling up the bandwidth. The

background transfers are usually made at the same priority as the foreground ones.

2.2 Multicast protocols

Deering's thesis on IP multicast ([53]) paved the way for one-to-many communication over the network infrastructure. IP multicast uses the the Class D range of IP addresses to represent multicast groups. The network is responsible for maintaining multicast relationships and making multiple datagram copies. Group membership and communication are handled by a protocol called the IGMP (Internet Group Management Protocol). IP multicast has seen only limited deployment: it is inhibited by its lack of scalability wrt the number of nodes in the group and the number of groups in the network. IP multicast defies the stateless nature of the IP network. Currently IP multicast is used in academic environments ([54]) and in private networks for media streaming and video conferencing applications.

Multicast at the network level is inherently unreliable. There have been many solutions to reliable multicast proposed in the past. These protocols are unscalable because they do not completely solve ACK flooding in the reverse path [55], [56]. Proposals for quasi reliable multicast have also been made to improve scalability [57].

2.2.1 Single Tree Overlay Multicast

End-system multicast [69] was one of the earliest proposed overlay multicast protocols. In Narada, end-systems self-organize into a delay-optimized overlay mesh and communicate using a distributed protocol. Narada is a good alternative to IP multicast since, apart from giving comparable latencies, it shows better scalability and lower overhead on routers. Narada is aimed only at small-sized groups and can scale to only tens of nodes. Narada constructs overlay multicast trees by first constructing a mesh. The mesh acts as the control path through which routing and control information is exchanged. Scalability is limited because end-systems maintain a complete group membership list. Data is transmitted over a spanning tree constructed from this mesh. Recently, a multi-tree version of ESM was proposed [76], [58] that masked the disadvantages of single-tree protocols.

Yoid ([68]) is an overlay multicast protocol proposed quite at the same time as Narada. Much on the lines of Narada, Yoid maintains a mesh for transmitting control information between participants and a tree for multicasting data. A joining node is aided by a rendezvous host for bootstrapping. Unlike Narada, each Yoid node is connected to a subset of the nodes in the network to maintain the mesh and the tree.

Overcast [86], a scalable single source overlay multicast routing protocol, is primarily designed for large-scale groups and bandwidth-intensive applications such as video streaming. Live content can be multicast through a self-organized topology. First, the protocol builds an efficient distribution tree taking into consideration the bandwidth available at each node. A new node joins the tree as far away from the root as possible without compromising on the

throughput. To avoid control overhead, the first few levels in the tree have a degree of one so that upon a root failure, another node can take charge. Since Overcast builds deep distribution trees, latency is traded off for throughput.

This decade saw the emergence of *structured* peer-to-peer[1] systems as a scalable extension to the traditional client-server approach. A Distributed Hash Table (DHT) is a distributed data structure used to build such systems. Scribe [71] is a multicast protocol that uses a DHT called Pastry[67]. Scribe uses Pastry's reliability, self-organization, and locality properties to support a scalable and fault-tolerant application-level multicast mechanism. Each group is associated with a Pastry key called *groupId*. A multicast tree comprises of all the paths from the node responsible for the *groupId* (the root) to the member nodes of the group. These paths are formed by the member node hashing (in effect, sending join messages) to the *groupId*. All non-member nodes that route this message become forwarders to this group. Multicast sources for a given group route their messages to the group's root node which then disseminates across the tree.

Though a lot of proposals for single-tree overlay multicast came after Narada and Yoid, the single important limitation of single tree approaches was still waiting to be tackled: Single trees have poor robustness to node failures. Failure of one node will lead to the entire subtree under it disconnected. Most of the nodes in a single-tree protocol are leaf nodes which make them non-contributors to the multicast, defying the notion of a balanced peer-to-peer system that can cater to bandwidth heterogeneity amongst the peers.

[1]Though peer-to-peer networks are a kind of overlay networks, in this thesis, the word 'overlay' is used synonymously with 'peer-to-peer'.

2.2.2 Multi-path Approaches

The failure of single tree approaches resulted in the multi-path approach where more than one path is used in transmitting packets to a particular node. One of the first proposals was Co-OpNet [59] which used multiple description coding to split packets into multiple streams and send them either directly via a powerful server or via the clients receiving the live stream. The coding ensured that the perceived video quality was proportional to the number of substreams received.

Bullet [70] splits the stream into multiple blocks and uses a single tree on top of a mesh. Nodes receive only a subset of the blocks from their parents in the tree, the remaining blocks retrieved from other nodes randomly chosen using a distributed algorithm called *RanSub*. Bullet however incurs a high control overhead due to this scheme of orthogonally retrieving packets.

Splitstream [72] is a multi-tree protocol that maintains multiple Scribe trees. For ease of explanation, we present Splitstream in Section 4.3 while presenting experiments comparing Chunkyspread with Splitstream.

Chainsaw [73] and Coolstreaming [61] are swarming-style data-driven multicast protocols that do away with trees to improve resilience. Each overlay node (proactively or reactively) notifies neighbors of data arrivals and employs a pull-based approach to retrieve blocks. Coolstreaming has been used in the Internet for TV broadcasts. These mesh protocols are a new point in the design space. They employ a data driven approach to multicast. Though they are more resilient than trees, they are known to suffer from greater control overhead and latencies. There have been hybrid designs ([92] and [91]) that utilize

the strengths of both trees and swarming protocols. In particular, mTreebone constructs a tree of stable nodes while employing a mesh-based multicast to the rest of the network. [90] compares mesh-based and multi-tree protocols and explores the tradeoffs between the two approaches.

[76] assessed the feasibility of overlay multicast protocols supporting large-scale live streaming applications by analyzing real-world Akamai traces; using these traces along with online and offline bandwidth measurements, they concluded that real-world hosts indeed have enough bandwidth to support themselves in most cases. [78] points out the limitations in the applicability of Scribe in heterogeneous environments especially with respect to its anycast and push-down operations.

Incentive-based p2p protocols try to enforce end-hosts to contribute resources. There have been many proposals in the literature that apply to file-sharing and streaming applications. Bittorrent [74] is a popular file-sharing protocol in widespread use that divides the file into multiple pieces and lets the peers download the pieces from one another. Peers employ a tit-for-tat mechanism to limit free-ridings the system. [77] adopts a taxation model on peer-to-peer streaming multicast applications to encourage resourceful peers to contribute bandwidth to the system and subsidize for the poor peers. [81] employs a credit-based technique on Splitstream to detect free-riders. According to this scheme, trees are reconstructed periodically so that each pair of neighbors gets opportunities to donate and receive between each other on successive reconstructions. The protocol does not fully answer how to tackle heterogeneity in the system.

2.2.3 Content Distribution

There are other kinds of content dissemination that blur the line between unicast and multicast. Peer-to-peer file sharing systems like PAST [63], Napster [64], Bittorrent [74] disseminate (or/and share) files across peers in the overlay network. While multicast protocols are real-time (the content is usually a sliding buffer of a live stream), these protocols do not have that requirement (the content is typically a large file). Akamai [65] is a vast content delivery network (CDN) that maintains replicas of content from web servers. Users are served by one of the nearby Akamai hosts. This helps in reducing the bandwidth load in the web servers that actually host the content and also in reducing the latencies at the user side.

CHAPTER 3

PRIORITY LAYERED APPROACH TO TRANSPORT FOR HIGH BANDWIDTH DELAY PRODUCT NETWORKS

3.1 Introduction

Long-haul optical links are enabling the deployment of high-speed enterprise networks that extend across large geographical distances. These networks are typically owned or leased by enterprises and organizations ([3], [28], [4]) and operated by ISPs providing VPN and managed network services ([5], [6], [7]) over the raw fiber with specific service level agreements (SLAs). These SLAs provide Quality-of-Service (QoS) guarantees that are implemented by DiffServ-enabled [26] routers.

The transport protocol performance issues seen on these long-fat pipes are not fully solved by existing solutions. XCP [30] for instance performs well, but it requires changes to routers. Other protocols ([31], [41], [35]) that rely on end-to-end signals for congestion control on the other hand do work with the general best-effort Internet. While these are more aggressive than TCP, they remain only cautiously aggressive in order not to sacrifice stability and fairness.

This thesis presents a new transport protocol, called Priority Layered Transport (PLT), that finds a sweet spot between the assumption of disruptive change to the infrastructure (XCP), and the assumption that only the best-effort Internet is available. PLT achieves performance that is consistently as good as or better than XCP, while not requiring any infrastructure support that is not already available to enterprises running over VPNs or leased optical channels. In

particular, PLT exploits the fact that strict priority queuing is available today on these networks through DiffServ. PLT improves upon the cautious approach of transport protocols; it achieves this by assigning packets from a given single transport flow into two strict priority classes. This results in two 'sub-flows', a high-priority flow and a low-priority flow. The high-priority flow, which runs at the same priority level as competing TCP flows (i.e. best effort), operates conservatively (AIMD). The low-priority flow, however, aggressively exploits spare capacity in the network. Because of the strict priority, the aggressive low-priority flow does not interfere with the conservative high-priority flow or legacy TCP flows. This isolation affords us considerable flexibility in the design of the aggressive low-priority component of the congestion control algorithm. We exploit this flexibility to design a transfer protocol that will do no worse than regular TCP, and will often do substantially better.

In short, PLT strikes a balance between fairness and aggressiveness. This balance derives from the fact that, on the high-priority flow, we provide the same fairness as TCP, while on the low-priority flow, we favor aggressiveness over fairness. Specifically, the low-priority flow is unfair over short time frames, though is fair over long time frames (Sections 3.3.4 and 3.4). As a result, a PLT flow never gets less throughput than it would have as a straight TCP flow, and a TCP flow competing with PLT flows never gets less throughput than it would have in a pure TCP environment. Because PLT is guaranteed to 'do no harm' to other high priority flows, PLT can afford to be much more aggressive on the low priority channel. It does not, for instance, have to worry about starvation. As a result, a PLT flow can get considerably more throughput than it would have as a TCP flow. We believe that, from an engineering standpoint, our balance of fairness and aggressiveness is a perfectly reasonable choice.

One consequence of better throughput is lower flow completion time. This is in fact more significantly manifested for short web transfers (mice) in PLT, since the low priority flow can send more data during TCP's conservative slow start, thereby completing the flows much faster than TCP.

Another benefit to our approach is that the low priority flow primarily complements the performance of the high priority flow; hence, PLT can be designed to have any congestion control algorithm for the high priority flow. Unless stated otherwise, we assume TCP as the high priority flow.

An immediate applicability of PLT can be seen in commercial VPNs; PLT can be deployed in performance-enhancing proxy (PEP) boxes on the edges serving end hosts that run on legacy TCP, so as to enhance the end-to-end performance (on similar lines to [23], [24], [25]). In the absence of priority queuing, however, PLT needs to stop sending packets at the low priority in order to avoid affecting other TCP flows. For this reason, PLT includes a mechanism to detect the absence of priority queuing and to disable the aggressive flow in response.

This work makes the following contributions:

1. We propose a new approach for transport over long, fat pipes that exploits existing strict priority queuing in routers to eliminate the constraints of conservativeness that have limited previous approaches.

2. We present the design of a specific transport protocol, PLT, that utilizes this approach. In particular, PLT uses TCP's legacy AIMD as the high-priority component and a more aggressive MIMD as the low-priority component.

3. We designed and implemented PLT on a user-space TCP stack ([27]). We

present the results of our emulation experiments on the Emulab [80] testbed. We show that PLT yields near-100% goodputs even in lossy long-fat pipes, produces low mice completion times, and can produce good utilization even in the face of changing network conditions.

4. We present the results of a thorough performance comparison of PLT with XCP, FastTCP, and TCP based on ns2 simulations. We show that PLT can yield at least 30% more goodput than XCP or FastTCP in scenarios with high cross-traffic and random losses and unlike FastTCP or XCP, can sustain near-100% utilization even when elephants (long flows) interact with a high rate of mice. Further, PLT never performs significantly worse than the other two protocols.

5. We present and evaluate an auto-discovery and shutdown mechanism of PLT's low priority flow when strict priority queuing is not implemented at the bottleneck. We also validate its operation over the wide-area public Internet.

6. We built a PEP based on our user-space implementation, thus demonstrating the feasibility of immediate deployment of PLT in VPN environments.

The chapter is organized as follows. Section 3.2 describes our protocol design and architecture in detail. Section 3.3 presents the results of a simulated comparison of PLT, FastTCP, XCP, and TCP. Section 3.4 presents the results of our emulation experiments.

3.2 PLT Design and Implementation

Transport protocols like TCP need to make a number of basic decisions such as when to send a packet (based on the congestion and receive windows), which

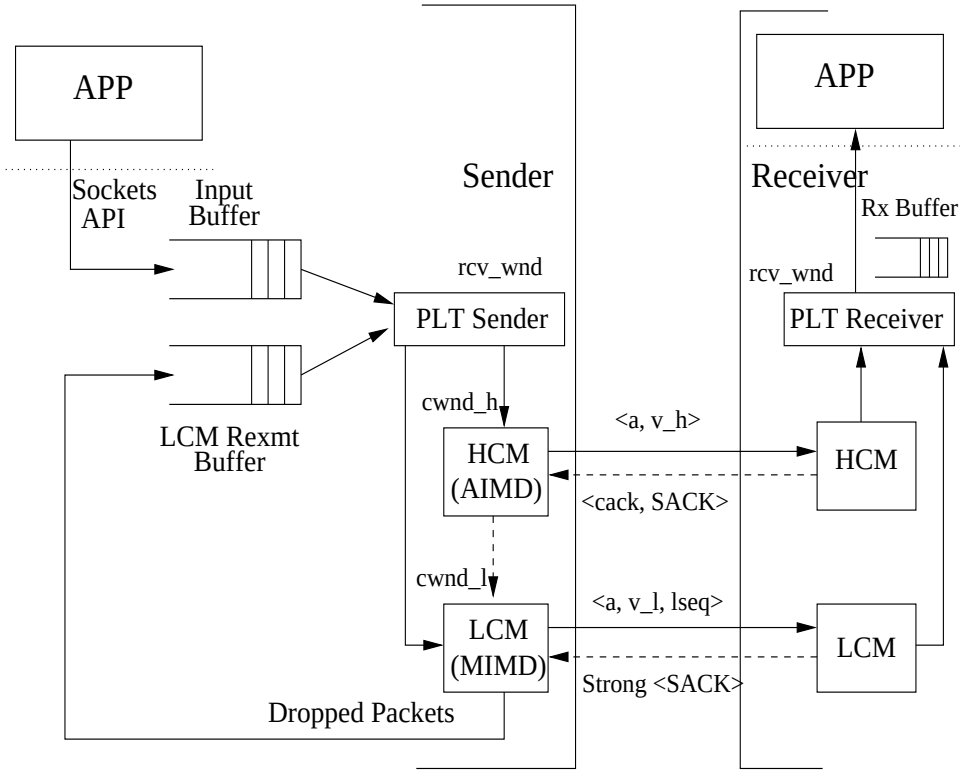


Figure 3.1: PLT Architecture

packet to send (a new packet or a previously sent packet), and which packet to release from the transmit buffer. In PLT, these decisions are challenged by the fact that any given packet can potentially be sent via one of two priority subflows, each with its own congestion control algorithm. Figure 3.1 shows the PLT architecture, including the major protocol modules, the data packet flow through those modules (solid lines) and some of the control information flows (dotted lines). In what follows, we walk through Figure 3.1 at a high level, followed by detailed descriptions of each module.

Application data arrives at the input buffer of the PLT sender, either directly via the sockets API if PLT is implemented in the OS stack of the application (shown in Figure 3.1), or via a TCP connection with the application host if PLT

is implemented as a PEP. Upon arrival in the input buffer, the bytes of the application data are assigned sequence numbers, called the *actual sequence number*, as with the normal TCP operation. The actual sequence number stays associated with the application data throughout the system all the way to the receiver buffer at the PLT receiver.

Ultimately, application data will be transmitted either via the conservative, high-priority channel (sub-flow), or via the aggressive, low-priority channel. These channels are managed by the High-priority Control Module (HCM) and the Low-priority Control Module (LCM) respectively. Both the HCM and the LCM maintain their own congestion windows. In addition, they also have their own *virtual sequence numbers* which are different from, but map into the actual sequence number. This is necessary because application data is interspersed between the two channels. Without virtual sequence number spaces, data acknowledgements would be highly inefficient. Furthermore, providing separate virtual sequence numbers for each channel allowed us to use a nearly unmodified TCP implementation for the HCM. Data packets sent from a CM (either the HCM or the LCM) to its corresponding receiver contain both the virtual sequence number assigned by the CM (v_h and v_l), and the actual sequence number (a). The acknowledgements, however, only needs to contain the virtual sequence numbers.

While the HCM and LCM individually limit the flow into their respective channels based on their congestion windows, it is the PLT sender module that keeps track of the total outstanding (sent but not yet acknowledged) bytes, and the receive window at the PLT receiver. This works as follows. The PLT sender knows, for each CM, the congestion window size and the number of outstand-

ing packets local to the CM. From this it can calculate the number of bytes that each CM can accept. The PLT tries to give each CM as many bytes as it can accept, limited by the total outstanding bytes and the total receive window. For instance, if a CM can accept 100K bytes, but the difference between the receive window and outstanding bytes is only 50K bytes, then the PLT sender will only give the CM 50K bytes.

The HCM runs standard TCP-SACK in its virtual sequence number space. Hence, it reliably transmits bytes assigned by the PLT sender to the HCM receiver. The same, however, is not true of the LCM. If the LCM determines that the LCM receiver has not received certain transmitted bytes, then it will enqueue those bytes back to the PLT sender's LCM retransmit buffer. This design choice was made because the LCM is inherently *lossy*, for instance, the channel can starve when the HCM traffic fully utilizes the bottleneck link. If the LCM is starved, it may not be able to deliver its bytes for a long time, and the overall flow will stall. If, on the other hand, the LCM feeds the lost bytes back to the PLT sender, it can subsequently deliver those bytes via the HCM.

3.2.1 PLT Sender

Given the above background, the operation of the PLT sender can be completely specified. As data arrives from the application to the PLT sender, the latter contacts the HCM and then the LCM to send packets. The PLT services the LCM retransmit buffer before the input buffer. The PLT always feeds the HCM before feeding the LCM. There is, however, one exception to the above. If the LCM retransmit buffer is not empty, and the HCM cannot accept more bytes

but the LCM can, the PLT sender may still not feed the LCM from the LCM retransmit buffer. The reasoning here is that the HCM is likely to be able to service the LCM retransmission buffer soon, with a smaller probability of yet another retransmission (discussed later in section 3.2.5).

The CM sender contacts the PLT sender when the former wants to choose a new packet to send, as long as the CMs congestion window and the overall outstanding window allow it to send the packet (as described above). The PLT sender would choose the next packet differently for the two CMs. For the HCM, it prioritizes sending lost LCM packets over packets not yet transmitted. As for the LCM, the PLT sender checks only the set of non-transmitted packets for the next packet to send. The LCM can make an exception by retransmitting a lost LCM packet if the PLT sender finds the HCM1 window to be very small when compared to the size of the LCM retransmit buffer. Once the CM decides to send a packet, it tags the header with the virtual sequence number of the packet and sends it to the receiver at the appropriate priority. The PLT sender maintains maps between the actual and the virtual sequence number spaces for the LCM and the HCM, and stores a mapping as and when the CM sends a packet. The PLT sender would erase the mapping when the CM receives an acknowledgement and informs the PLT about it.

As stated above, the HCM operates according to TCP-SACK, so we do not need to further describe its operation. The next few sections explain the functioning of the LCM.

3.2.2 LCM Design

The LCM design comprises of the aggressive congestion control algorithm and a simple acknowledgement mechanism.

Acknowledgment scheme at the LCM

Because the LCM channel in itself is not fully reliable, there is no notion of a cumulative ACK as TCP has. Instead, the LCM receiver sends a stream of ACKs containing contiguous SACK blocks to the LCM that are treated as strong positive and negative acknowledgments. The NACK'ed bytes are then queued into the LCM retransmit buffer.

To prevent ACK drops, we need a mechanism by which the LCM receiver knows whether or not to retransmit an ACK. For this, the LCM maintains a sliding window within which it can still usefully receive ACKs (bytes which it has neither re-queued nor deleted). It conveys the left edge of this window to the LCM receiver via a sequence number lseq (see Figure 3.1). The LCM receiver cycles through the SACK blocks within this window with each consecutive ACK.

To ensure that the reverse traffic does not affect other high-priority traffic, the LCM receiver usually sends ACKs at the low priority. To ensure progress and avoid starvation of the low priority ACKs due to congestion in the reverse path, the receiver occasionally sends an LCM receiver ACK at the high priority. As these ACKs are small and relatively infrequent, we found from our experiments that this does not increase the overhead of the high priority channel significantly.

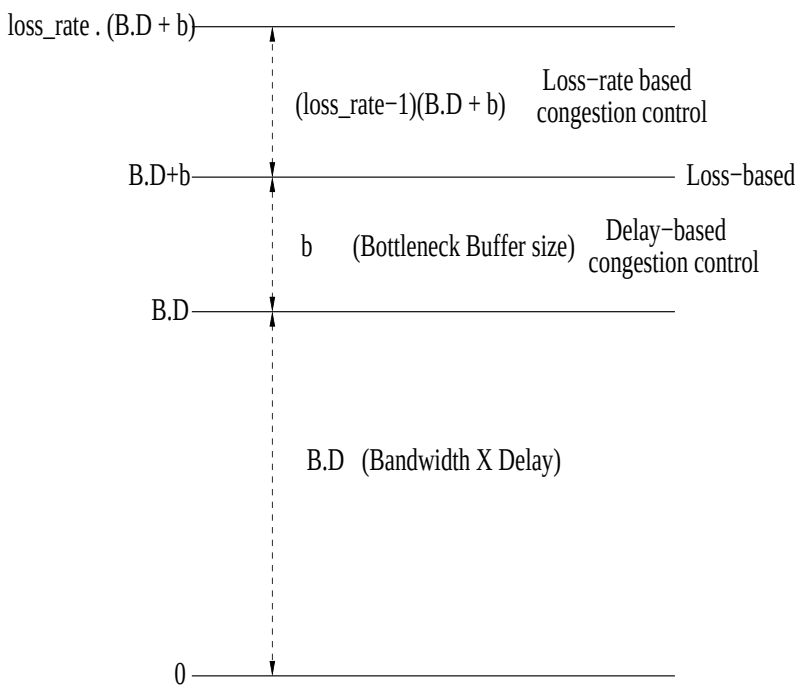


Figure 3.2: Regions in which the congestion control algorithms operate

Congestion Control in the LCM

The LCM contributes to the aggressive behavior of PLT through its congestion control. But, unlike traditional congestion control schemes, the requirements for the LCM are different; it needs a congestion control algorithm that is more aggressive even if that results in relaxed guarantees on stability and fairness; an MIMD approach with a small decrease factor seems a good fit. To further enhance LCM's robustness to temporary traffic bursts and random losses, we adopt a *loss-rate-based* control (similar to [40]), as against a loss-based or a delay-based approach. A congestion control scheme based on *loss rates* ramps up as long as losses are within some finite threshold; this gives an additional leeway for the protocol to protect itself from cutting down during temporary network congestion and random losses.

According to this approach, PLT maintains a *target loss rate* μ, a threshold at which the LCM operates. Time is divided into epochs, each epoch roughly spanning the RTT of the flow (calculated using the existing approach in TCP). At the end of each epoch, the LCM calculates the loss rate (p) during that epoch as the *ratio* of the number of packets lost in transit to the total number of packets for which the receiver had responded to. The overall loss rate (P) is then calculated as an exponential moving average. The LCM then reassigns its congestion window (cwnd) as:

$$cwnd = \begin{cases} \alpha.n + cwnd & \text{if P} < \mu \\ \beta.cwnd & \text{otherwise} \end{cases}$$

where α is the increase factor, β is the decrease factor, and n is the number of packets acknowledged by the receiver in the epoch.

The congestion window decreases also when there is a timeout. At the LCM, timeouts usually occur when the higher priority traffic congests the bottleneck and queues start filling up. Since such occurrences are likely, we introduce another cutback parameter γ which is greater than β so as to reduce LCM windows faster when HCM is saturating the pipe.

During a sustained saturation of HCM traffic at the bottleneck, the LCM will time out till its congestion window falls below one. Then the LCM probes periodically so as to ensure the LCM's congestion window can quickly fill up the spare bandwidth when it becomes available. As a more explicit measure, the HCM notifies the LCM whenever the former has cut back. The LCM, checks if its congestion window is below one and if it is, sets back the window to some initial value (say, a fraction of the slow start threshold of the HCM if it is running TCP). We also observed that there may be a case when the HCM wants to open up its congestion window but is constrained by the outstanding window.

In that case, the HCM forces the LCM to decrease its congestion window. These are the only two cases when the HCM interacts with the LCM (see Figure 3.1).

Other LCM details

Burst Control: Since LCM's congestion control alters its congestion window periodically, packets may be sent in bursts resulting in losses. To avoid these losses, the LCM is equipped with a burst control module based on a simple ACK-pacing mechanism much on the lines of FastTCP[31]. The burst control module also restricts the LCM from sending more than a certain number of packets at the same time.

Dependence on target loss rate: The functioning of PLT is dependent on the parameter μ. We set a value that is large enough to detect losses and cater to a range of loss rates where the HCM, namely the TCP-SACK misbehaves. In most of our simulations and in the implementation, we set default values of 5% and 0.5%. It is easy to see that any sustained loss beyond this value will lead to successive cutbacks of the LCM window leading to a drop in the flow's goodput.

Choosing a very high value for μ would enable the LCM to function well over a large range of loss rates. But the loss rate threshold is the percentage of losses that the LCM is willing to tolerate at the gain of being more aggressive with the channel and avoid cutbacks during congestion bursts. Using an arbitrarily large value of the threshold can result in wastage of bandwidth at those non-bottleneck links before the bottleneck link in the source-receiver path. Further, a high loss rate threshold results in maintaining large outstanding windows since the system needs to tolerate a larger loss. This is also not desirable.

A more practical alternative is to adaptively alter the loss rate threshold based on the moving average loss rate observed by the flow. The flow reverts back to the default value when the loss rates decrease. Let us note that the value of μ is crucial only during cases with high and sustained random losses (in the form of cross traffic bursts or channel losses). And in such cases, PLT can do well with this simple μ tuner. In this work, however, we set μ statically.

3.2.3 The PLT Receiver

The PLT receiver chooses to send an acknowledgement with the respective virtual sequence numbers. For this reason, the HCM-r and the LCM-r each maintain an outstanding window (usually a bitmap) for their respective virtual number spaces, apart from the main outstanding window and a receive buffer maintained by the PLT receiver. The HCMr is exactly the TCP-SACK receiver and the HCM ACK is identical to an acknowledgement packet of TCP-SACK as shown in Figure 1. The LCM-r, however, needs to just send some number of SACK blocks (restricted by the TCP header) for the LCM sender to know the packets received and sent. These SACK blocks are contiguous and cycle through the outstanding window of the LCM-r. We note that the LCM-r cannot advance its outstanding window unless it knows explicitly that the LCM sender has advanced and scheduled the lost packets to the retransmit buffer. Hence the LCM sender may have to tag the TCP packet with another field indicating the leftmost virtual sequence number of the LCM packet whose receive status is unknown at the LCM sender(denoted *lseq* in Figure 3.1).

The receiver sends one ACK for every received packet and uses the virtual

sequence number space. To ensure that the reverse traffic does not affect the other connections, the receiver usually sends ACKs for the low-priority packets at the low priority. To ensure progress and avoid starvation of the low priority packets due to possible congestion in the reverse path, the receiver every now and then, sends an LCM-r ACK at a high priority. This does increase the overhead at the high priority channel but that overhead is quite low considering the small length of ACK packets. In fact, to further reduce the low priority ACK overhead, the receiver can choose to use the delayed ACK mechanism (to send an ACK for every n low priority packets received).

3.2.4 Implementation details

The PLT architecture discussed so far is based on an implementation that we built over a user-space TCP stack called the Daytona [27]. We have tried to modularize the HCM as much as possible so that in the future, any TCP-friendly congestion protocol can be potentially used at the HCM. Apart from the data buffers, the PLT sender also maintains a couple of hash tables to map the virtual and the actual sequence numbers.

Starting a PLT connection is simple: the sender and the receiver confirm through a SYN exchange whether PLT is enabled: this field is present in the byte space provided for optional fields in the TCP header called the TCP options. In the VPN deployment scenario, in which a host may often know in advance that the receiver is PLT-enabled, these additional options may not be necessary.

We had also observed that the PLT sender attaches extra fields in its packet headers. These fields are packed in TCP options. The HCM payload should

contain the virtual sequence number. The LCM payload should tag along the virtual sequence number as well as *lseq* as we had described earlier. While the HCM ACK is similar to the legacy TCP ACK, LCM ACK simply has the SACK blocks specified in the TCP options. Because of space restrictions ([51]), the LCM payload cannot specify timestamps and can piggyback along with an LCM ACK only if there are no SACK blocks that need to be sent. The priority is specified in the IP header ([52]) using the six most significant bits of the DiffServ field called as the Differentiated Services Code Point (DSCP).

PLT Shutdown: In situations where priority queuing has not been enabled on routers, PLT's aggressive low priority traffic should be stopped. We have designed a mechanism whereby lack of priority queuing can be detected and LCM can be disabled. We use the fact that during a period of congestion, when HCM packets get dropped, no LCM packet can get past the bottleneck queue if strict priority queuing is enabled. The PLT receiver periodically monitors the loss percentages on both the channels to make a decision. The receiver could err if the packet drops are caused by non-bottleneck cross traffic or random losses: in such cases, both the HCM and the LCM could face non-zero packet losses. PLT should not be disabled then. In our implementation, if the receiver senses that the HCM has started to lose packets, and the LCM is seeing "significant" non-100% losses (that is, if its loss percentage is greater than typical network losses but less than 100%), then the receiver disables PLT and informs the sender about it. In our implementation, the receiver uses the loss threshold to make this decision.

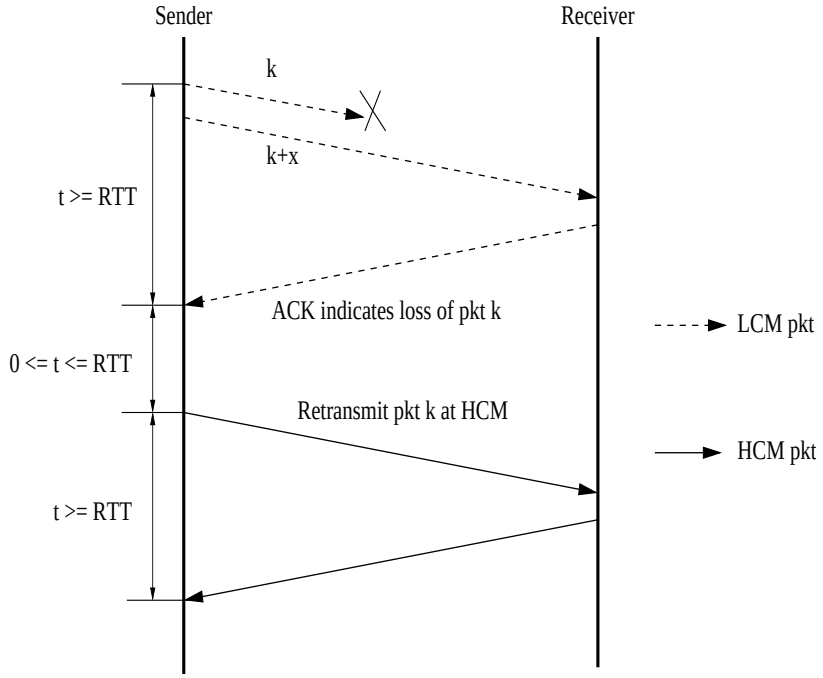


Figure 3.3: A sample timeline between a PLT sender and PLT receiver to indicate the upper limit for the receiver window bounds.

3.2.5 Receive Window Bounds

We had indicated earlier that the receive window could be the bottleneck because of PLT's aggressive nature. In this section, we come up with theoretical bounds for receive window sizes required to ensure a smooth functioning of PLT.

Let us consider a single flow in a network with bottleneck bandwidth b and a round-trip delay of τ, with no external source of packet loss. Let us also assume that all LCM retransmissions are done at the HCM. In our implementation description earlier in this section, we had mentioned that the HCM retransmits lost LCM packets. Suppose an LCM packet gets lost and the sender gets to know about the loss through LCM SACKs (instead of a timeout). Since LCM

packets are sent only after HCM saturates its congestion window, the former would require a time interval of between 0 and $1.\tau$ after the sender knows about the packet loss, for the HCM to react to it (see Figure 3.3). Let us assume that the HCM has a large enough congestion window at that time to send all the lost LCM packets within the next round trip. This means that any packet initially sent and lost at the LCM will take between τ and $2.\tau$ for the HCM to retransmit the packet. Considering that one more RTT is consumed for the retransmission, a receive window (call it *rwnd*) size of at least $3.b.\tau$ is required to sustain the protocol.

$$rwnd \geq 3.b.\tau \tag{3.1}$$

The minimum receive window size that PLT requires depends on the duration of an LCM timeout. Note that the LCM timeout usually occurs when LCM packets starve in the bottleneck as the HCM packets fill the pipe. If the duration of a timeout is $n.\tau$ where (n > 0), then,

$$(n+1).b.\tau \leq rwnd \leq (n+2).b.\tau \tag{3.2}$$

to not let the HCM get stuck because of limited window size. This condition holds, if we assume that the HCM congestion window during the timeout is large enough to send all the timed out packets in the same RTT.

The assumption made above that all the packets lost (or timed out) at the LCM can be fit in the HCM's congestion window may not hold all the time. In such cases, the HCM may have to take more than 1 round trip to complete. Since the LCM may be sending packets during that time, it could expand the outstanding window. To prevent this situation, the LCM can chip in by sending the remaining data. But since the LCM is lossy, the frequency of retransmissions

in this channel is greater than in the HCM. Increase in the number of retransmissions further increases the outstanding window size. Hence, it is useful to determine the lower bound of the HCM congestion window for which the aforementioned assumption holds.

Let C_h and C_l be the HCM and LCM congestion windows at the time of the LCM's packet losses. Let B be the bottleneck buffer size; then the maximum congestion window size that the HCM can achieve (before a packet loss) is $b.\tau + B$. Let l be the number of packets lost among the LCM packets sent in the previous RTT. Let us assume that the HCM is in the congestion avoidance phase and that the packets are solely due to congestion losses. Then, for the assumptions to hold,

$$C_h + 2 \geq l \tag{3.3}$$

$$l = C_h + C_l - (b.\tau + B) \tag{3.4}$$

Then,

$$C_h + 2 \geq C_h + C_l - (b.\tau + B) \tag{3.5}$$

Hence,

$$C_l \leq (b.\tau + B) + 2 \tag{3.6}$$

Let the maximum value of C_l be denoted $maxC_l$. When the congestion window of HCM is C_h,

$$maxC_l = \frac{(b.\tau + B) - C_h}{1 - \mu} \tag{3.7}$$

where μ is the loss threshold. This follows from the way μ was defined:

$$maxC_l \;=\; ((b.\tau + B) - C_h) \;+\; \mu.maxC_l \tag{3.8}$$

For condition 3.6 to be satisfied for all values of C_l,

$$maxC_l \leq (b.\tau + B) + 2 \tag{3.9}$$

Therefore, we get

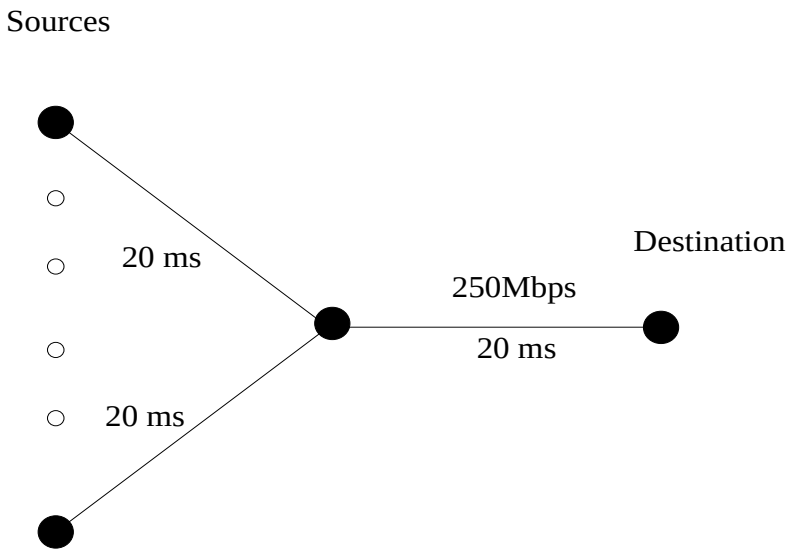
$$C_h \geq \mu.(b.\tau + B + 2) - 2 \tag{3.10}$$

This is a rather weak bound for HCM's congestion window above which lost LCM packets can *always* be sent via HCM within one RTT, hence restricting the outstanding window to the aforementioned value. From the relation, we see that choosing lower μ values decreases the possibility of requiring huge outstanding window sizes.

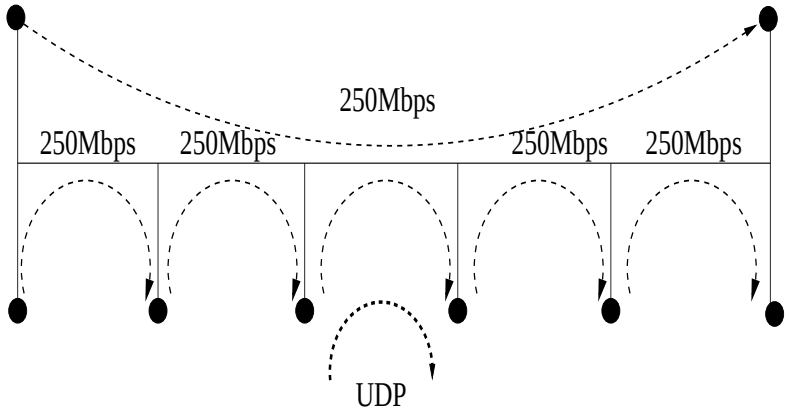
In the presence of significant random losses and transient congestion, HCM's window sizes are small and the LCM chooses to retransmit itself; it can be shown that the receiver window sizes are still under control. We also note that as the number of flows increases in the network, the maximum receiver window size required for each flow reduces roughly by a factor of the number of flows.

3.3 Evaluation: Simulation study

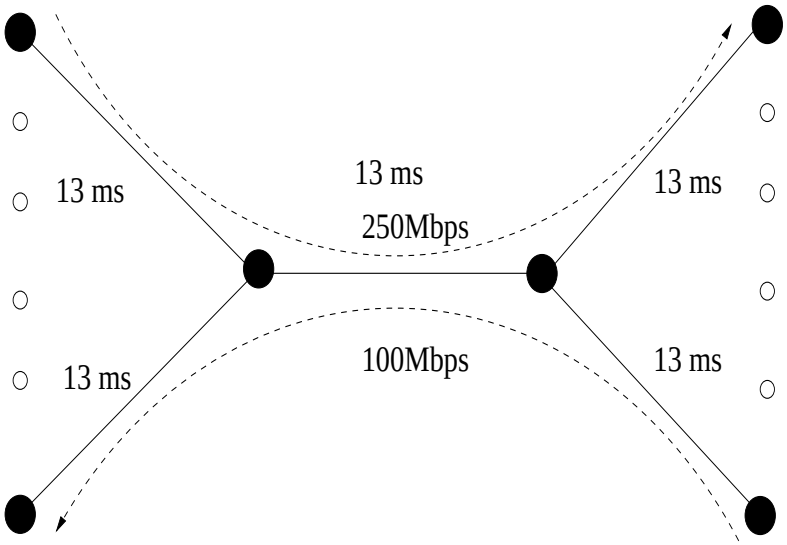
This section presents an ns-2 comparison study of PLT in high bandwidth environments against FAST and XCP. Unless specified otherwise, the HCM in all our simulations is TCP-SACK. The simulation code corresponds to the implementation described in the previous section.



(a) Single bottleneck topology



(b) Multi-bottleneck topology



(c) Single bottleneck topology

Figure 3.4: Topologies used in the simulations

3.3.1 Experimental parameters

The first table above lists the default parameters in our simulations. The subsequent table gives a key of the protocol versions used in our plots.

The single bottleneck experiments are run with round-trip times of 80ms, as shown in Figure 3.4(a). We have experimented with two sets of buffer sizes: one at 10% of the bandwidth-delay product and the other exactly at the bandwidth-delay product. The small buffer experiments are shown owing to the increasing importance in restricting buffer sizes of the routers ([50]) especially in long fat networks. The contribution of the LCM to the overall performance is more significant in such cases since the HCM (TCP-SACK) functions poorly with small buffers in long fat pipes.

Parameter	**Policy/ Value**
Increase factor α	0.2
Decrease factor β	0.95
Timeout decrease factor γ	0.75
Bottleneck bandwidth (BBW)	250 Mbps
Max. Receiver Window	3.BW.RTT
Router Drop Policy	Tail drop
Epoch Duration	RTT
Payload size	1000B
Initial LCM Window size	0.1*BW.RTT

Key in plots	Explanation
PLT(x)	PLT with μ=x
PLT-U(x)	PLT with window size=20.BW.RTT
PLT-Short(x)	PLT(x) with three levels of priorities
PLT-N-ACK	LCM ACKs sent at high prty @ 1 for every N ACKs
FAST(x)	FAST with α=x
FAST-MI(x)	FAST with MI option

The default receiver window size is set to thrice the bandwidth delay product; such a restricted size is a bottleneck when there is a single flow in the network, since the timeout period is set to a little larger than that value. For such cases, we have also tested PLT with a receiver window size of 20 times the bandwidth-delay product, and is denoted PLT-U.

We have also conducted experiments on a topology with multiple bottleneck links (Figure 3.4(b)) each with a bottleneck of 250Mbps and one of the links experiencing UDP cross traffic, with the other arrows in the figure indicating the transport protocol flows (PLT, XCP, FAST, or TCP). All results are recorded after the system reaches a steady state.

We predominantly work with the default μ value of 0.05 for reasons mentioned earlier. We also present some results with other threshold values, for a better understanding of the effect of changing μ. A PLT experiment using a loss rate threshold of x is denoted PLT(x) in the plots. There are two other versions of PLT that we evaluate in some of our experiments for more insight. PLT-Short (see the table above) is a version of PLT that uses three layers of priority instead of just two. In this version, LCM packets sent during the slow start phase of the

HCM are assigned a greater priority when compared to the usual LCM traffic. This will guarantee better completion times for short flows amidst competing LCM traffic. PLT-N-ACK is a version of PLT-Short with 1 LCM-r ACKs sent at high priority for every N LCM-r ACKs (the default value in PLT being 5 ACKs). This version is particularly useful while evaluating the effect of reverse traffic.

All our experiments compare PLT primarily against FAST and XCP. We also present TCP-SACK experiments to show the performance improvement of PLT due to the use of LCM.

While XCP and TCP-SACK work well with their default parameters, FAST depends on the value of α, the target queuing delay parameter used in its congestion control. We observed in our simulations that the value of this parameter is very crucial to its performance. In particular, a system of n flows needs at least a buffer size of $2n\alpha$ packets for it to converge well. We have used the suggested value of 100 in our simulations. The performance of FAST may improve in some cases if the parameter α is tuned carefully.

In addition to α, there is another feature in the FAST protocol called the multiplicative increase (MI) option. When it is set, FAST multiplicatively increases as long as the queuing delay at the bottleneck does not reach a certain MI threshold, after which FAST uses its regular congestion control. We find that both the versions of FAST, with and without MI options, have issues: FAST with MI option (denoted FAST-MI), is able to grab the channel faster than the regular FAST and hence more suited to short and bursty applications. However, FAST-MI results in frequent oscillations if the bottleneck queue size is small. We consider FAST-MI in our simulations only if mice are present in the system.

3.3.2 Effect of random losses

Figures 3.5(a) and 3.5(b) show the goodputs of a single flow with an infinite source and a single bottleneck link (Figure 3.4(a)) for increasing random loss rates with both large and small bottleneck buffers respectively. PLT yields very high goodputs even at non-zero loss rates, whereas FAST and XCP yield lower goodputs at those rates.

With zero loss rates, the window evolution of the HCM (TCP-SACK) shown in Figure 3.6(a) is well-known. The LCM window increases during those times when the HCM is not filling the bottleneck queue, but backs off otherwise. The LCM can expect to send packets successfully as long as the bottleneck is not saturated by HCM packets. But when the bottleneck queue starts getting filled up by the HCM packets, the LCM packets cannot make it to the receiver. Hence, the LCM times out and cuts back successively till either the LCM window size goes below one or the bottleneck gets under-utilized again (potentially due to a HCM cutback). This pattern of LCM window is shown in Figure 3.6(b). TCP-SACK works well with large bottleneck buffers and no random losses (Figure 3.5(a)) in the network.

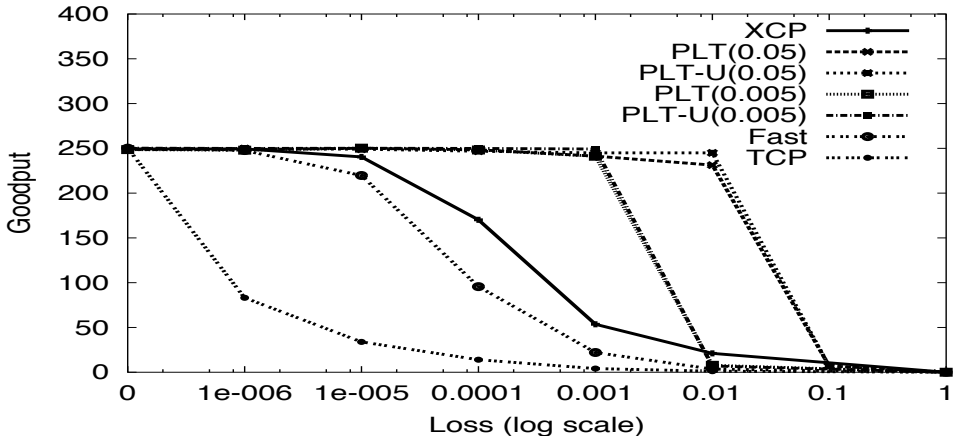
With non-zero loss rates (Figures 3.5(a) and 3.5(b)), the HCM never fills the queue, and so the LCM kicks in and utilizes the available capacity. The high goodput is maintained as long as the loss rate is on the average less than the loss rate threshold. This is the reason for setting the loss threshold at a conservative value of 5%. An alternative approach is to adaptively change the loss threshold based on the observed loss rate. The reason behind some loss of goodput even at loss rates less than the loss rate threshold is the fact that when losses occur, the outstanding window frequently hits the maximum receiver window size, hence

acting as a bottleneck. To corroborate this fact, we find that PLT-U delivers near-100% goodputs for non-zero loss rates as well.

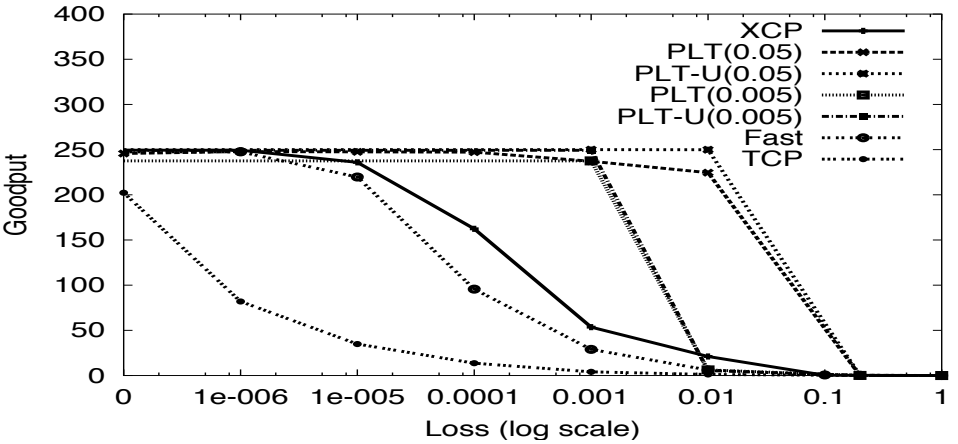
Figures 3.5(a) and 3.5(b) also show the goodputs of FAST(100) and XCP with increasing loss rates. With zero loss rates, both FAST(100) and XCP yield perfect goodputs (as expected), but with increasing loss rates, they yield very low goodputs since their congestion control algorithms lead to frequent and drastic cutbacks at such high random losses. We also find that the size of the bottleneck has almost no effect on the goodputs of the two protocols. The goodput figures for TCP-SACK in both the cases are as expected.

3.3.3 Effect of number of flows

The next three graphs, Figures 3.7(a), 3.7(b) and 3.7(c), show the performance of the protocols with 10, 50 and 100 flows respectively sharing the same bottleneck, as a function of increasing bottleneck queue size. We observe that on increasing the number of flows, PLT can yield superior aggregate goodputs, close to 100%. XCP also yields very high goodputs while FAST is sensitive to the value of α: hence for a value of 100, FAST is more prone to oscillations as the number of flows increases. We see that FAST shows reduced bottleneck utilization (ranging from 75% to 93%) in the graphs though the reduction is less with greater number of flows. TCP-SACK does quite well with large buffer sizes which shows that there is negligible contribution from the LCM in the corresponding PLT plot. However, its performance with small buffers and large number of flows predictably suffers.



(a) Goodput vs. Loss rate for large buffers

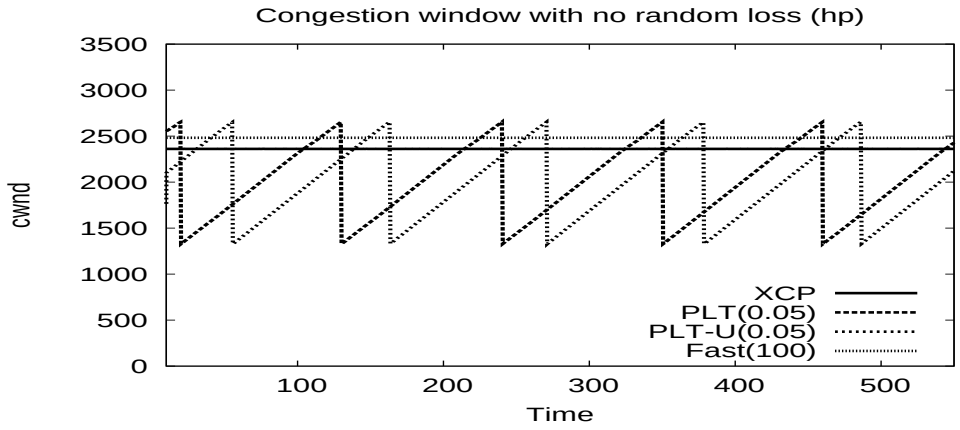


(b) Goodput vs. Loss rate for small buffers

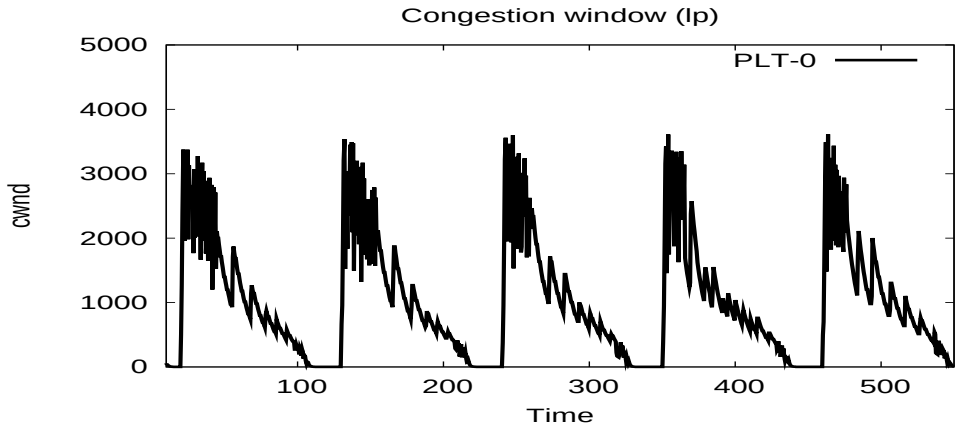
Figure 3.5: Single flow Single bottleneck experiment

3.3.4 Fairness

It is easy to observe that PLT flows are at least as fair to each other as the constituent HCM flows are. In the network scenarios that we simulated, we observed that a single connection dominates the low priority queue between the times the low priority window goes below one. However, the connection that dominates in each such cycle varies at random thus providing fairness to all the connections in the long term. For example, with 10 flows and bottleneck



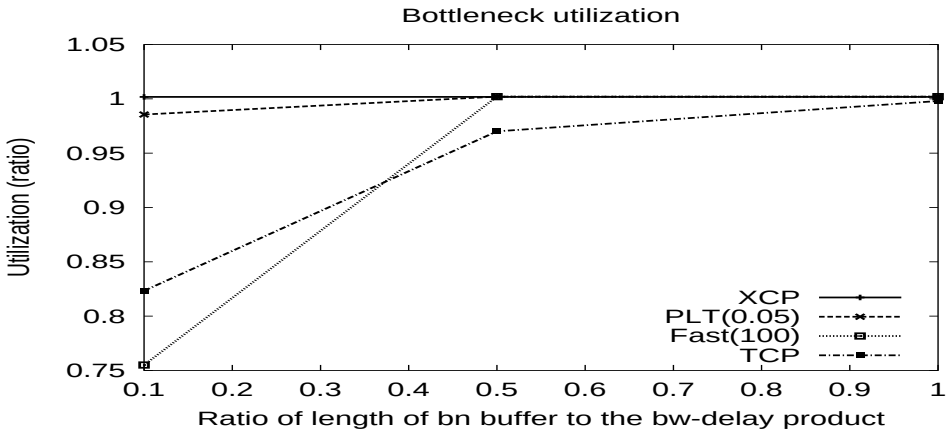
(a) High priority cwnd:No loss



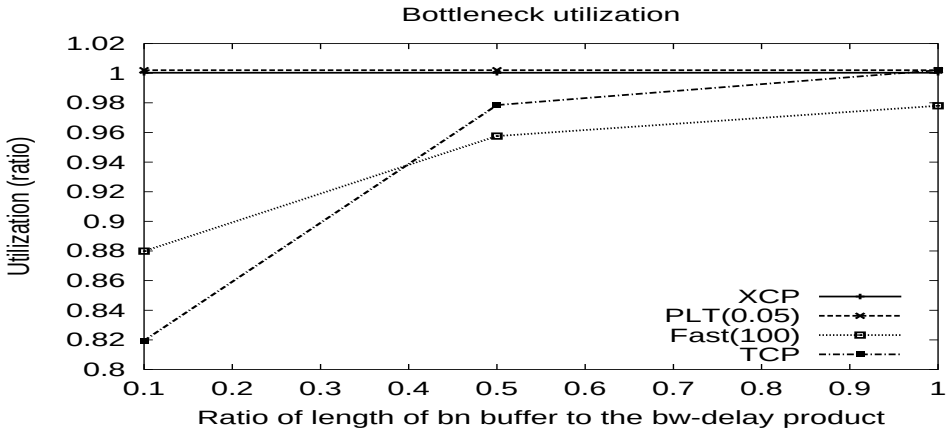
(b) Low priority cwnd:No loss

Figure 3.6: The congestion window evolution with no losses.

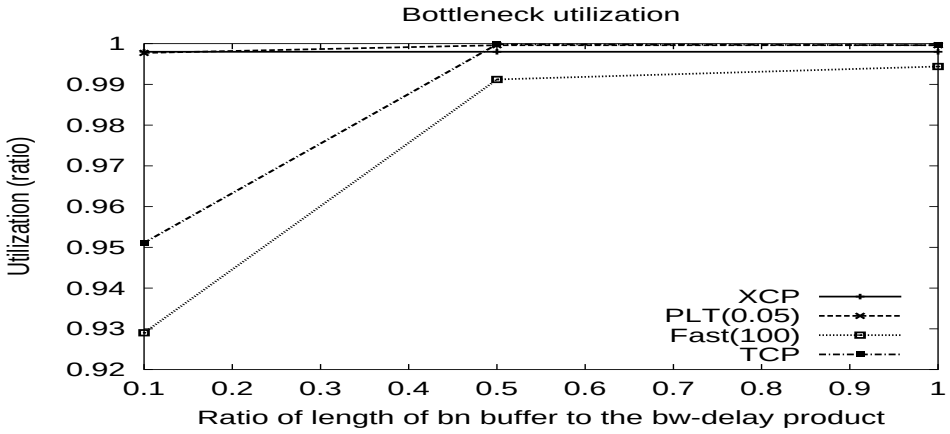
buffer sizes of 10% of the bandwidth-delay products, where TCP-SACK at the HCM behaves poorly, the average goodputs of the PLT flows is 23.9Mbps with a minimum goodput of 22.93Mbps and a maximum goodput of 27.43 Mbps.



(a) Bottleneck Utilization with 10 Flows



(b) Bottleneck Utilization with 50 Flows



(c) Bottleneck Utilization with 100 Flows

Figure 3.7: Effect of increasing number of flows in the network

3.3.5 Mice completion

We tested the completion time of flows sharing a common bottleneck arriving at a steady Poisson rate of 500 flows per second. The volume of the flows is pareto distributed with an average size of 25 packets as is the case with Internet flows. Figure 3.8(a) shows the frequency distribution of the flow completion times, with the flows arranged in increasing order of size. We find that more than 95% of the PLT flows complete (the lowest curve in the graph) within a couple of RTTs (160ms). While PLT's good performance here is due to the fact that the initial congestion window of the LCM is high enough to fill the pipe, we note that it is the fact that LCM packets run at lower priority that allows us to set the initial window size aggressively. We have set the initial window size to be 10% of the bandwidth delay product though the burst control module would not let that many bytes be sent at the same time. We also observe that XCP's completion time curve almost coincides with that of FAST-MI and TCP; in these cases the flows take at least 3 RTTs to complete. Fast without MI takes at least one more RTT to complete. As the mice rate increases to the point of saturating the pipe, PLT converges to the performance of TCP-SACK.

3.3.6 Mice and elephants

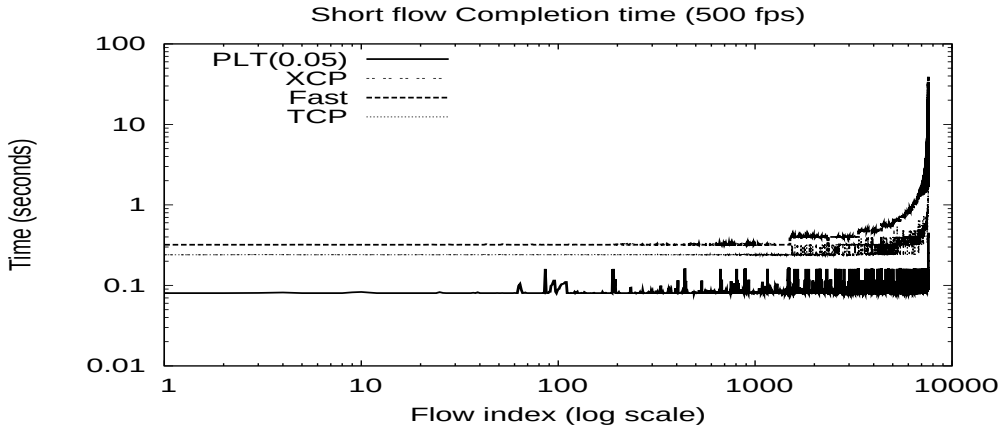
We studied the effect of varying mice rates in a network consisting of 50 elephant flows on a single bottleneck topology. The mice sizes are distributed the same way as in the previous experiment. Figures 3.8(b) and 3.8(c) show how PLT fares when a number of mice flows and elephants interact with each other. Figure 3.8(c) shows the average completion times of flows with increasing mice

rates. The graph also shows the results of *PLT-Short*, a version of PLT that uses three layers of priority instead of just two. In this version, LCM packets sent during the slow start phase of the HCM are assigned a greater priority when compared to the usual LCM traffic. This will guarantee better completion times for short flows amidst competing LCM traffic. On the whole, PLT's mice complete much faster than both FAST and XCP as shown in the figure by 1 to 3 RTTs. The average bottleneck utilization during the mice arrivals, is plotted in 3.8(b). The graph shows that PLT can sustain very good utilizations of greater than 97% even at high mice arrival rates.

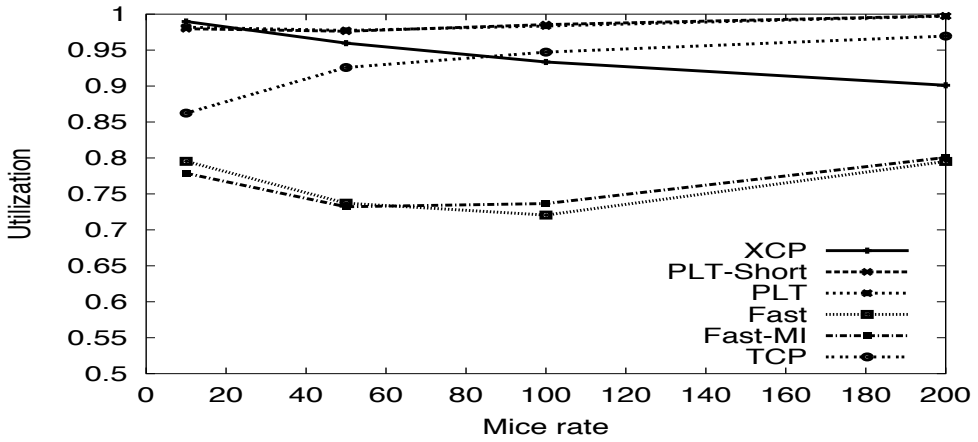
We also observe that while XCP shows a small but a noticeable dip (by up to 10%) in the bottleneck utilization as the mice rates increase, FAST shows very poor utilization (of just around 72%). This can be possibly improved by a more careful choice of α. FAST also suffers from poor average mice completion times, of the order of 1 second, thanks to a considerable number of outliers which yield very high completion times. FAST-MI, on the other hand, predictably shows much better completion times, of the order of 500ms.

3.3.7 Effect of cross traffic on a multiple bottleneck topology

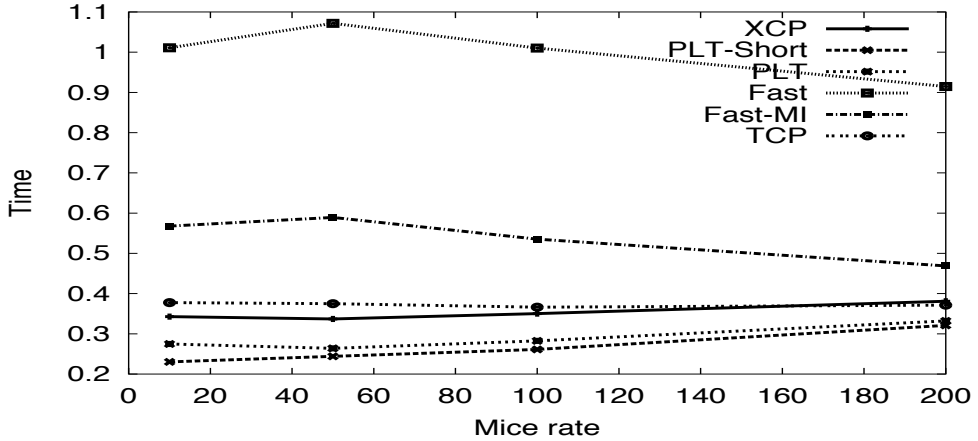
With the multiple bottleneck topology (Figure 3.4(b)), we introduce bursty exponential on-off UDP traffic at average idle and active durations of 50 ms. The UDP traffic competes with the protocol flows hence resulting in cutbacks in their congestion windows and decrease in goodput. The links have bandwidth capacities of 250Mbps and only the central link is subject to cross traffic. Each bottleneck has 3 flows passing over it while there are 3 flows passing along the



(a) Mice completion time (500fps)



(b) Bottleneck Utilization with mice and elephants



(c) Completion time (in seconds) with mice and elephants

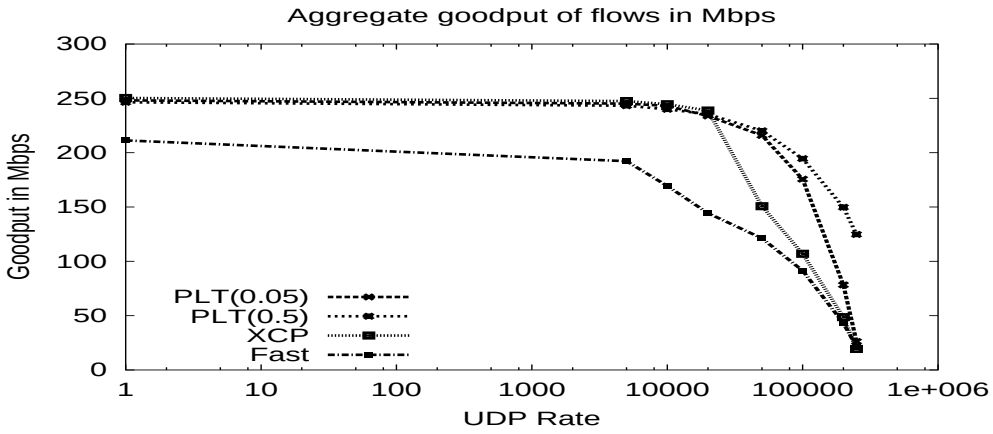
Figure 3.8: Effect of short flows in the network

entire length of the path.

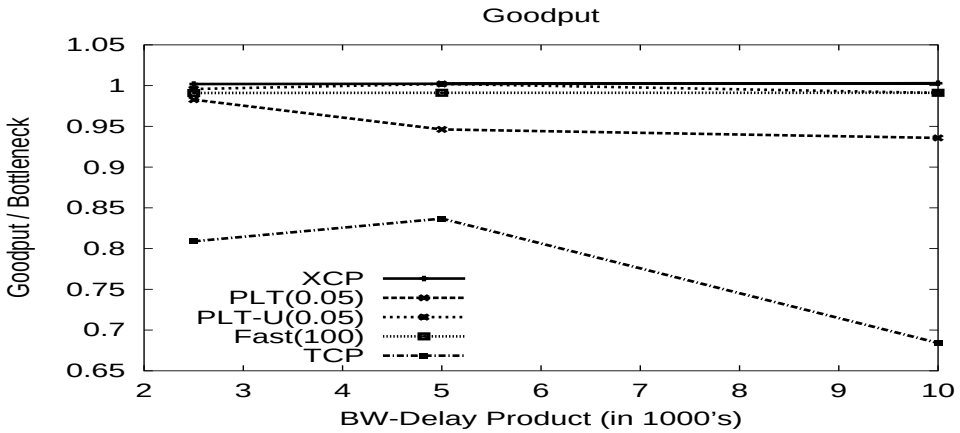
We observe the aggregate goodputs from the flows passing through the bottleneck, for various rates of the UDP traffic in Figure 3.9(a). It shows that even with non-zero UDP rates, PLT shows very high aggregate goodputs. XCP also performs well, but at higher UDP rates (50-100 Mbps) PLT(0.05) does better than XCP. For example, with 50Mbps bursts, PLT shows an aggregate goodputs of 225Mbps (90%) and a bottleneck utilization of 100% while XCP shows aggregate goodputs of 150Mbps.

3.3.8 Increasing bandwidth-delay product

Figure 3.9(b) shows goodputs of PLT(0.05), XCP, and FAST (100) with increasing bandwidth delay product. The bottleneck buffer sizes are set to 10% of the respective bandwidth delay product. We find that both XCP and FAST fare quite well even at products as high as 10000 packets. We also observed in our simulations that FAST-MI (not shown in the plot) resulted in poor goodputs between 70 and 80% of the product, showing that the performance of FAST-MI is sensitive to the values of the MI threshold and α. PLT shows reduced goodputs of around 95% because of receiver window constraints. This can be observed from the fact that at higher receiver window sizes, PLT-U can yield superior goodputs.



(a) Goodputs vs. cross traffic on the parking lot topology

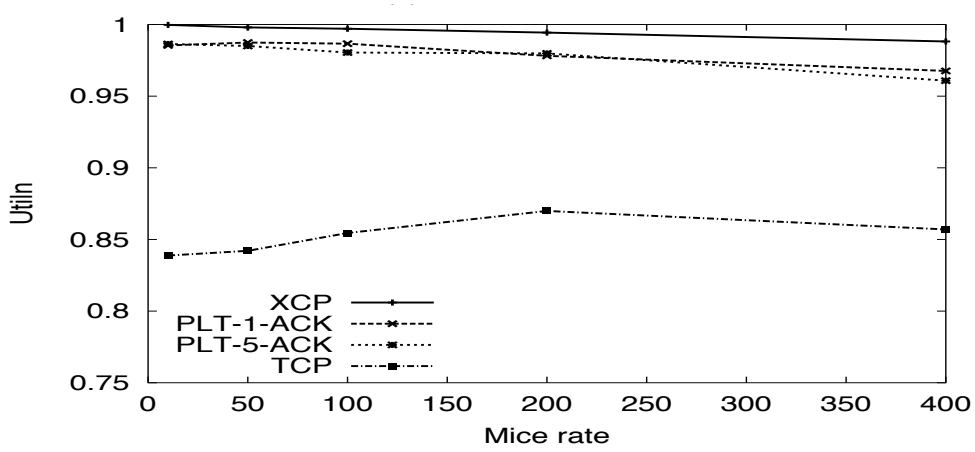


(b) Effect of Increasing N/W capacity

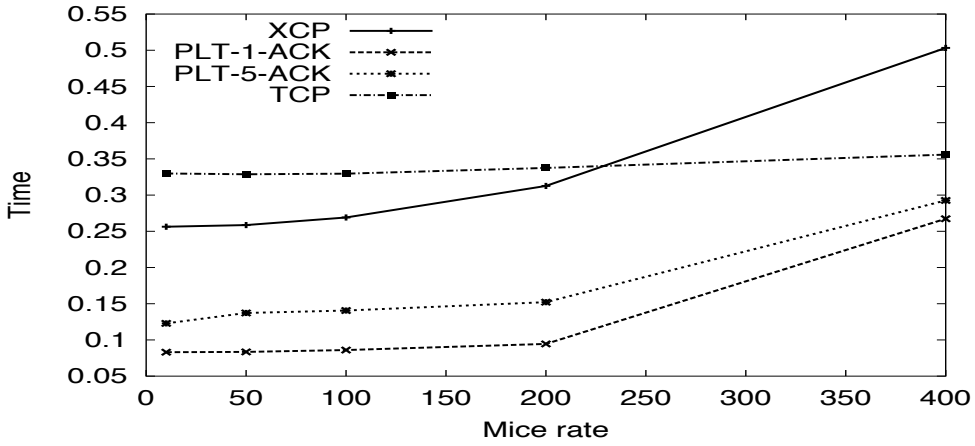
Figure 3.9: Effect of cross traffic and network capacity

3.3.9 Effect of Reverse Traffic

To observe the effect of reverse traffic in flows, we established 50 elephants sharing a single bottleneck with the reverse path populated by a steady rate of mice. The topology is a dumbbell topology (Figure 3.4(c)) with a bottleneck bandwidth of 250Mbps and a reverse bottleneck bandwidth of 100Mbps. Figures 3.10(a) and 3.10(b) show the forward path bottleneck utilization and the mice completion times of XCP and PLT. For simplicity, we have considered only PLT-



(a) Bottleneck utilization



(b) Average mice completion

Figure 3.10: Reverse traffic case

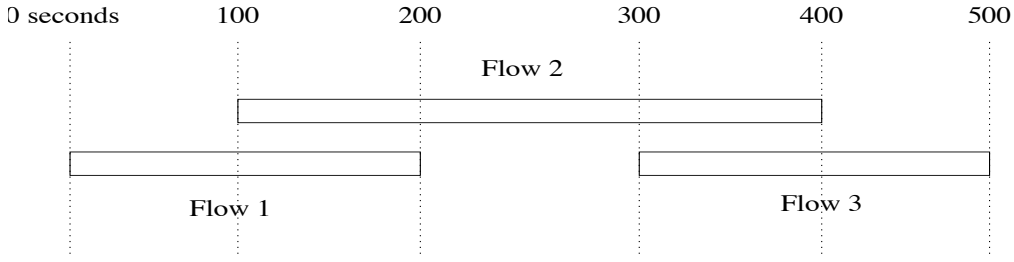
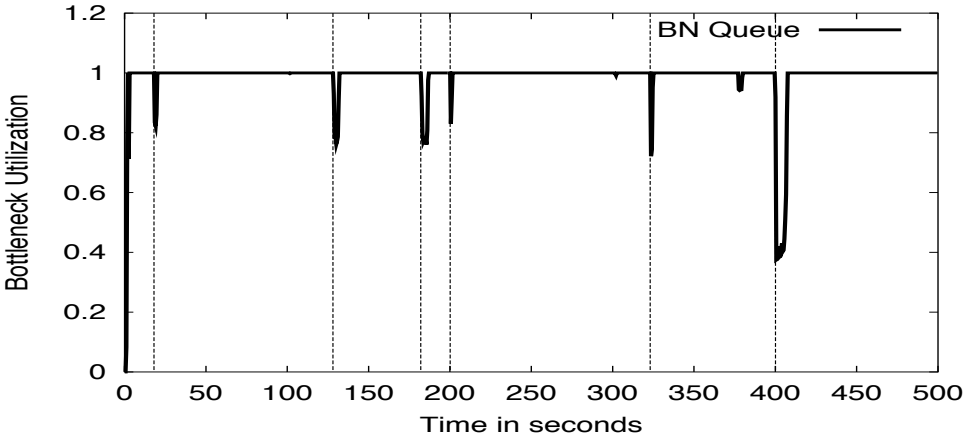
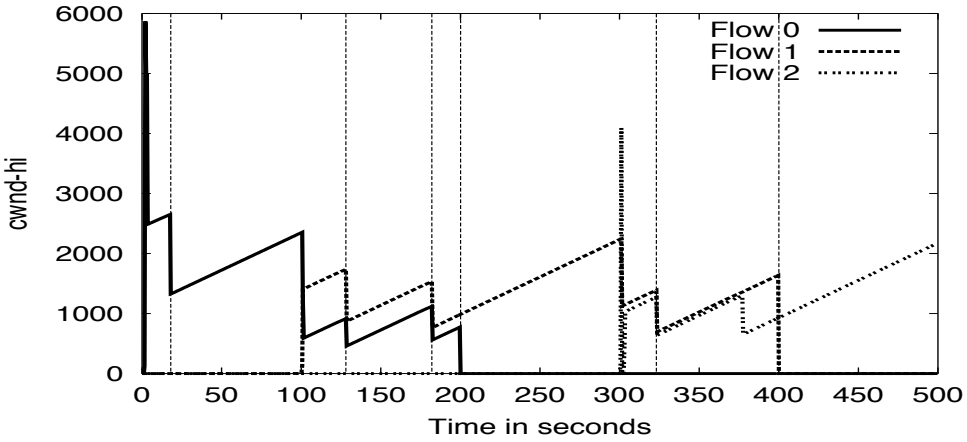


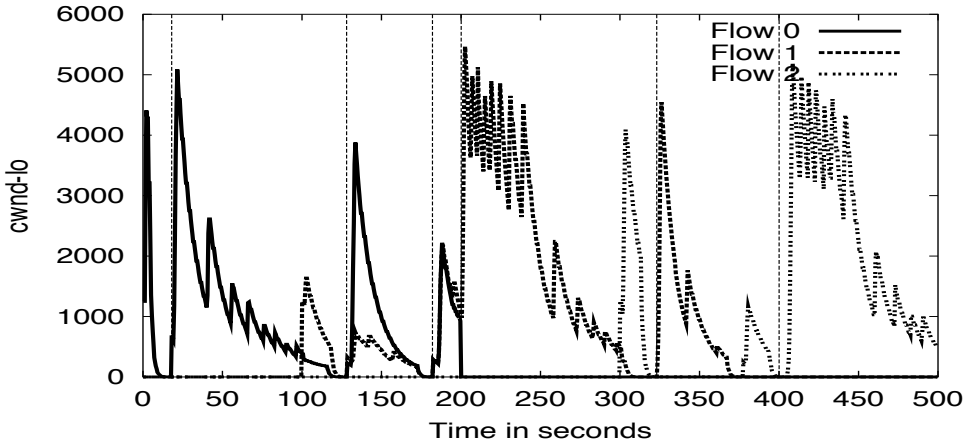
Figure 3.11: Dynamic scenario



(a) Utilization in dynamics



(b) HCM cwnd in dynamics



(c) LCM cwnd in dynamics

Figure 3.12: Results from the dynamic scenario

Short in this set of results, since we know the benefits of PLT-Short from the previous scenarios.

The LCM is not robust to reverse traffic if all the LCM ACKs are sent at the lower priority: a reverse path with sustained congestion would starve the LCM ACKs hence resulting in a large number of retransmissions, showing a performance similar to TCP-SACK. However, this can be improved by the receiver sending an LCM ACK periodically at the high priority. The graphs show three versions of PLT: PLT-1-ACK, PLT-5-ACK, and PLT-200-ACK which set this period to every 1, 5, and 200 ACKs sent at the low priority respectively.

We observe from the graphs that XCP elephants show high bottleneck utilizations even with high mice rates; notably, with 400 flows per second at which rate, the reverse pipe almost gets filled up, XCP shows only a small reduction of utilization (by 1.5%). Similarly, PLTs utilizations are high, but decrease at 400 flows per second with decreasing frequency of LCM ACKs at the high priority. TCPs utilization is only around 85%.

The mice completion times are on expected lines as well. PLT-1-ACK mice are able to complete faster since they receive LCM ACKs earlier (on an average) than PLT-5-ACK or PLT-200-ACK. XCP shows slightly higher completion times than PLT. The completion times expectedly increase with increasing mice rates: while PLT converges close to the TCP completion times, XCPs completion times also increase due to the increasing time taken by the routers to apportion the bandwidth to the flows. TCPs completion times remain more or less constant even at high mice rates because the reverse path does not get congested in the first place, thanks to the poor bottleneck utilization that the elephants suffer.

3.3.10 Effect of dynamic behavior

We take a closer look at PLT's performance in the presence of dynamics in the system. We take the case of the single bottleneck topology (Figure 3.4(a)) and three PLT flows arriving and leaving the network as shown in figure 3.11. Figure 3.12(a) shows the bottleneck utilization over this period of time. The average utilization over the running time is around 96%.

To understand Figure 3.12(a), it is necessary to observe the congestion window evolutions at the HCM (figure 3.12(b)) and at the LCM (figure 3.12(c)). There are six significant valleys seen in the graph roughly at time instants 20, 125, 180, 200, 320, and 400 seconds. These time instants are marked by vertical bars in the graphs. Let us denote these valleys 1, 2, 3, 4, 5, and 6 respectively. Valleys 4 and 6 occur because of the flows 1 and 2 ending respectively. We observe that the low priority module ramps up quickly (Figure 3.12(c)). Valleys 1, 2, and 5 happen when the newly joined flows 1, 2, and 3 perform slow start and subsequently make multiple cutbacks at the HCM (3.12(b)) till the utilization goes below 100%. The LCM traffic kicks in fast and fills up the pipe in no time. Valley 3 is an interesting one. Just before flow 1 leaves there is a HCM cutback in both the flows 1 and 2: the LCM traffic of both the flows fills up the buffer. But soon after that, flow 1 ends resulting in valley 4. This scenario shows that PLT can indeed fill up the spare bandwidth quickly in the face of changing network conditions.

#flows	TCP-TCP(s)	PLT-TCP(s)	PLT-TCP(l)
2	102.36-101.55	144.46-101.54	125.89-124.36
10	20.53-20.49	25.03- 20.12	25.05-25.02
50	4.13-4.15	5.28- 4.14	5.01-5.01

Figure 3.13: Average flow goodput: 's' denotes small buffers; 'l' denotes large buffers

3.3.11 TCP friendliness

There is a negligible interaction between high priority and low priority traffic if a high-priority packet arrives at the router just when a low priority packet is being transmitted. But we found out through experiments that the effect of this small interference is not seen in any of the results. Figure 3.13 tabulates the average goodputs of each flow when a number of PLT flows are run along with an equal number of TCP flows sharing the same bottleneck link in the topology shown in 2. The suffix s indicates small buffers and l refers to the case of large bottleneck buffers. For example, the third column in the second row shows that the average goodput of 5 PLT flows when run with large bottleneck buffers, along with 5 TCP flows is 25.05Mbps and that of the 5 TCP flows is 25.02 Mbps.

Fasts friendliness with TCP-Reno is scenario-dependent and though it has been theoretically proven that an equilibrium fair share exists, simulations still suggest that reaching that equilibrium may not be easy. XCP requires dynamic weighted fair queuing at the routers for it to be TCP-friendly.

3.3.12 HCM protocol independence

We have already mentioned that potentially any congestion algorithm can be incorporated into the HCM. To illustrate this fact, we have also conducted simulation experiments with PLT-Fast, where the HCM is Fast-TCP. We have gotten positive results so far with the protocol. For example, PLT-Fast(0.05) with large receiver windows, yielded near-100% goodputs in the single flow-single bottleneck scenario for loss rates less than 0.05. One of the main concerns with Fast-PLT however is that Fast is more aggressive than TCP, hence can exert a lot more tension on the outstanding window than the TCP-Sacks version of PLT.

3.4 Evaluation: Implementation

We have evaluated our user-level implementation of PLT on various topologies on Emulab ([80]). To configure priority queuing in the FreeBSD and the Linux boxes, we used the ALTQ and the tc tools respectively. We tested with a bottleneck bandwidth of 155 Mbps and a non-bottleneck bandwidth of 622 Mbps, which correspond to the capacities of OC-3 and OC-12 carriers respectively (see Figure 3.14). The RTT is configured to 100 ms unless otherwise specified, hence the bandwidth delay product is nearly 2000 KB. Two sets of buffer sizes 300 KB and 2000 KB have been tested at the bottleneck. We used a simple file transfer application on top of the user-space stack for most of our experiments.

Random losses and mice completion: Our first two experiments study mice completion times and the effect of random losses, much along the lines of the simulation experiments. Figure 3.15(a) shows the goodput of a single PLT flow

with small buffers under various random loss rates. The graph shows that PLT yields 94-95% goodput as long as loss rates are less than the threshold. We also note that the overhead due to packet headers cost an additional 3.6% of the bandwidth. We have compared the protocol with TCP in the presence of large buffers, with expected results.

We next study the completion times of web transfers running on PLT flows. The web server services 100 flows per second over a period of 1 minute. The wall clock time difference between the connect and the close calls at the client is taken as the flow completion time. We plot the frequency distribution of the 90^{th} percentile of flow completion times for pareto-distributed flow sizes, in Figure 3.15(b). The completion times are on expected lines with PLT performing much better than TCP by saving at least 1-2 RTTs. Unlike in the simulations, the mice completion times shown here for PLT also include the TCP-SYN handshake and the TCP-FIN exchanges.

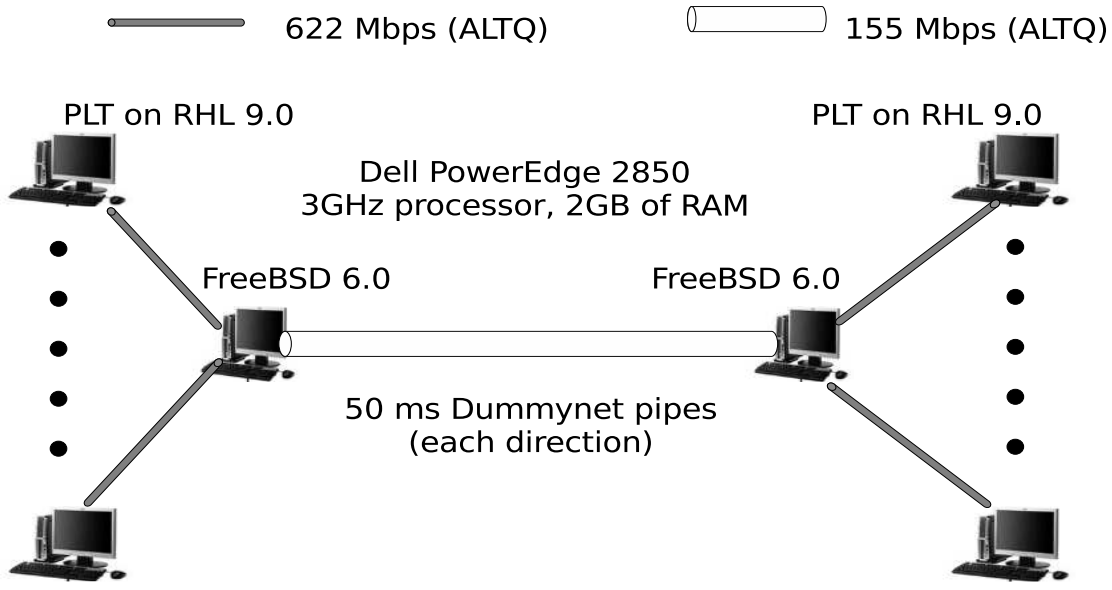
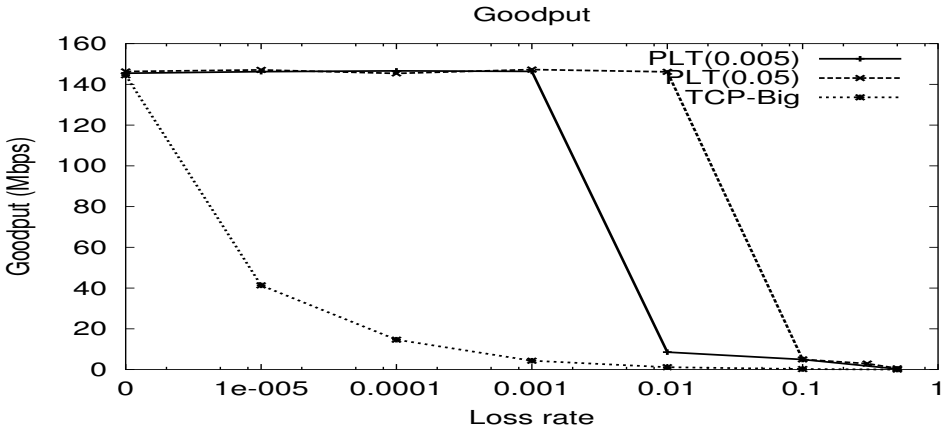
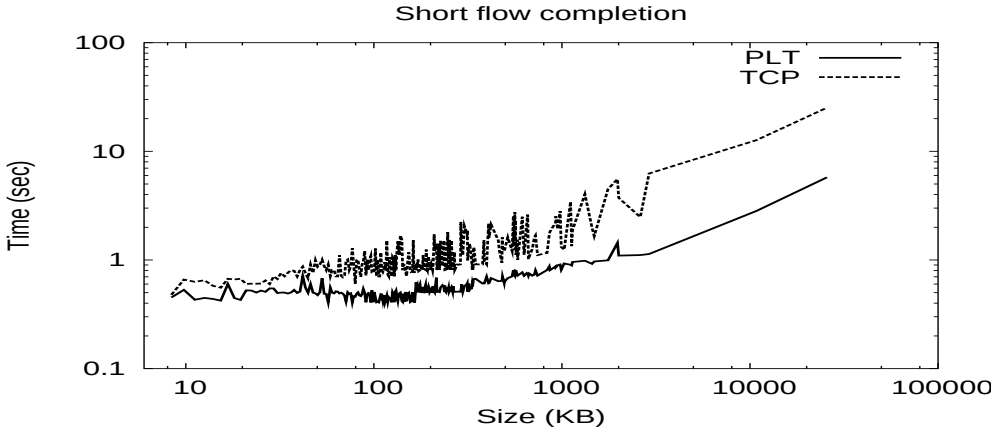


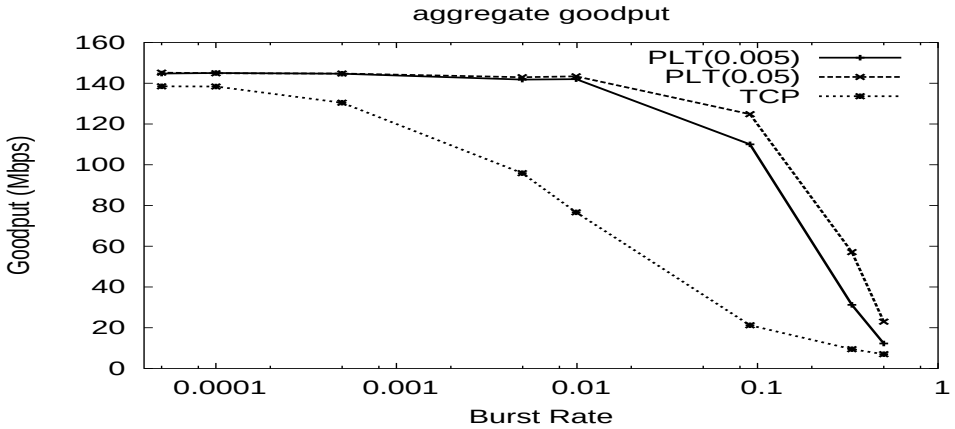
Figure 3.14: Implementation setup



(a) Effect of Random losses



(b) Mice completion time



(c) UDP Cross traffic

Figure 3.15: The important simulation experiments shown on Emulab

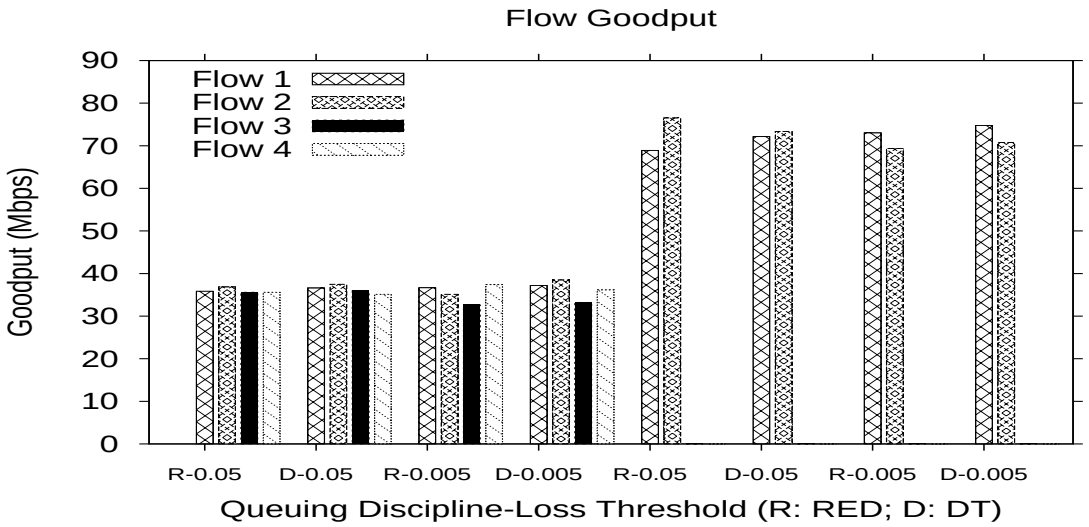


Figure 3.16: Fairness

	Loss	**Med**	10^{th}	90^{th}	**Err %**
No PQ	0.0	1.56	1.55	7.62	0
	10^{-5}	1.64	1.56	8.51	0
	10^{-4}	1.63	1.56	2.75	0
	10^{-3}	2.78	2.72	8.56	0
	0.01	16.27	2.97	31.75	0
	0.1	23.90	3.71	90.95	0
PQ	0.0	-	-	-	0
	10^{-5}	-	-	-	0
	10^{-4}	-	-	-	0
	10^{-3}	-	-	-	0
	0.01	430.62	183.49	479.58	40
	0.1	16.79	3.72	71.01	100

Figure 3.17: PLT-Shutdown

Effect of Cross Traffic Bursts: The graph in Figure 3.15(c) shows the aggregate goodputs of 4 flows on a dumbbell topology in Figure 3.14 with different rates of cross traffic. An exponential on-off UDP burst was sent through the bottleneck in both the priorities. Each burst averages the bottleneck capacity of 155Mbps (on each priority level) and lasts for 50 ms. The graph shows the aggregate goodputs with varying mean idle times (of the burst). The burst ratio on the x axis is the ratio of the burst time (50 ms) to the sum of the burst and the mean idle times in each cycle. We find that the PLT flows show near-perfect goodputs in the aggregate even with sustained bursts of 1%. TCP's performance in small bottleneck buffers is as expected.

LCM Fairness: As mentioned in Section 3.3.4, PLT is able to provide coarse-grained fairness with respect to the LCM traffic. Figure 3.16 shows the individual goodputs of PLT flows. Each flow is a large file transfer between a server and a client. The graph shows flow goodputs as a function of the number of flows, the queueing discipline (drop-tail or RED), and the loss-rate threshold. We observe that even if the LCM traffic is significant (around 20-30%), PLT is able to provide a fair share of goodput across flows in the network. In our experiments, fairness does not seem to be affected by changing the queuing disciplines or the loss rate threshold. However, with RED, we observed that the LCMs of the constituent flows showed a more fine-grained degree of fairness than with drop-tail queuing.

Performance Enhancing Proxy: We built a PLT-proxy that serves legacy TCP end-hosts and deployed it over a topology with two subnets connected through a path of routers that enable priority queuing (as could be configured on a VPN). The end hosts transparently maintain TCP connections between each

other, while the proxies transform the TCP connection into a PLT connection between the proxies that sit on the edges of each subnet. The end hosts get superior goodputs (95%) when compared to running TCP connections between each other, even at finite loss rates. Because the proxy is a user-space implementation, the system could scale only to 80-100 Mbps. We are in the process of implementing a kernel version of PLT.

PLT-Shutdown: We have conducted preliminary experiments on our PLT-Shutdown protocol; we tested a client downloading a 10 GB file from a file server application running over PLT, on a network subject to random losses. The loss threshold is 5%. Table 3.17 shows the median, 10^{th} and 90^{th} percentile times taken by the receiver to detect the lack of priority queuing and disable PLT, over 10 runs. We observe that when there is no priority queuing, the receiver disables PLT within a maximum of 8 seconds into the flow's inception when there is no external source of loss in the network. The detection time increases as the random loss rate increases, since random losses dominate over congestion losses as loss probability increases.

When priority queuing is enabled in the network, PLT-Shutdown does not disable throughout the flow's lifetime for low random loss rates. At 1% loss rate, PLT-Shutdown shows 40% false positives while at 10% loss rates, it shows 100% false positives. The reason for this happening is the same as in the previous case: random losses dominate over congestion losses.

We also validated this protocol by running PLT between a university in the east coast and a node in the Emulab testbed separated by a round trip of 70 ms and predictably, the protocol turned off within 1.01 seconds (at the median) into its inception.

CHAPTER 4

CHUNKYSPREAD: HETEROGENEOUS UNSTRUCTURED TREE-BASED PEER-TO-PEER MULTICAST

4.1 Introduction

With the recent emergence of a number of P2P IPTV startup companies (such as [58], [60]), P2P multicast has again become a hot topic. Many of these products are based on a 'data-driven' or 'swarming' style of multicast similar to Chainsaw [73] or Coolstreaming [61]. In the swarming approach, each overlay node advertises to its neighbors which packets (or blocks of packets) it has received, and the neighbors explicitly request blocks as needed. This approach is in contrast with the more traditional 'tree-based' style, whereby one or more delivery trees are defined by the overlay nodes, and packets are delivered along the trees without any explicit requests.

The primary stated advantages of the swarming approach are its simplicity and its robustness. The simplicity stems from the fact that that it requires no 'complex' distributed algorithm to build trees. The robustness stems from the fact that any neighbor can be called upon to contribute blocks of data, so the loss of any given neighbor does not cause a discontinuity in data delivery. These benefits, however, come at a cost: there is a basic tradeoff between *control overhead* and *delay*. This tradeoff is easy to see. Imagine that, in order to minimize delay, each node informed its neighbors as soon as it received any given packet so that those neighbors could request that packet as soon as possible. This clearly results in a considerable overhead. To reduce this overhead, each node instead waits until it has received some number of packets, and then ad-

vertises a bit-map indicating which packets it has. The longer the node waits, the more packets it can efficiently advertise in its bit-map. Coolstreaming [61], for instance, encodes 60 seconds worth of packets in its bit-map.

Tree-based approaches don't exhibit this control-overhead-versus-delay tradeoff. Rather, they have a *continuity-versus-delay* tradeoff. Namely, if a node's parent in the tree crashes or leaves the tree, then the node won't receive packets until it can find a new parent. If the node wishes to continuously play out packets to the application, then it must buffer enough packets to bridge the gap. The faster the tree-building algorithm can discover and fix a broken tree, the smaller this buffer has to be. Tree-based approaches can mitigate the effect of broken trees by constructing multiple trees and transmitting redundant FEC codes over some of them, which results in an *overhead-versus-delay* trade-off.

In this work, we present a new tree-based multicast algorithm, called Chunkyspread, that is far simpler than previous tree-based algorithms. Like swarming approaches, Chunkyspread is unstructured. Like the DHT-based Splitstream [72], Chunkyspread uses multiple trees to balance load among nodes, and indeed exhibits far more control over load than Splitstream. Chunkyspread reacts quickly to membership changes, scales well, and has low overhead. Furthermore, Chunkyspread is designed such that it provides a framework for adding new performance optimizations and constraints, such as tit-for-tat.

We believe that the relative simplicity and good performance of Chunkyspread makes it a viable candidate for P2P multicast. While ultimately this means detailed and thorough comparisons with swarming approaches[1] this

[1]Future works after Chunkyspread [90] have explored these tradeoffs and have made a systematic comparison between tree-based and mesh-based approaches.

work focuses on comparison with Splitstream. Doing so allows us to answer the question of finding the best tree-based P2P multicast algorithm, thus paving the way for later comparisons with swarming approaches.

This work makes the following contributions:

1. We give a detailed description of Chunkyspread[2], the first unstructured P2P multicast algorithm with fine-grained control over load.
2. We present a thorough simulation analysis of Chunkyspread's load control, latency optimization, responsiveness, and overhead.
3. Using the MSPastry simulation of Splitstream, we present an analysis of Splitstream over the same metrics, and compare Splitstream with Chunkyspread.
4. Again through simulation, we present preliminary and limited analysis of Chunkyspread for tit-for-tat, and for the basic trade-off of buffer size, data redundancy, and packet loss in the face of churn.
5. We present limited results of a complete implementation of Chunkyspread running on Emulab. These results validate our simulation results.

This chapter is organized as follows. Section 4.2 describes our approach in detail. Section 4.3 gives an overview of the existing multi-tree approach, namely Splitstream. Section 4.4 presents evaluations of both Chunkyspread and Splitstream.

[2]In [62], we presented an overview of Chunkyspread and some preliminary simulation results.

4.2 Protocol description

We start with a high-level overview of Chunkyspread, followed by a detailed description of its various components.

Chunkyspread constructs a single-source multicast group among a set of member end-systems. In other words, there is one sender, (which we call the *true source*), and multiple receivers. If the application requires multiple senders, then either multiple groups must be formed, or the multiple senders must first send to a designated single sender acting as the true source, which then transmits to the multicast group. Our implementation does not currently provide this latter capability nor does it provide a capability to change the true source in the middle of a multicast, although doing so would be relatively straightforward.

Like Splitstream, the true source transmits the multicast stream as M distinct slices. Each set of these M slices is said to constitute a *block* of stream. Each slice is transmitted over a separate multicast tree. But, quite unlike Splitstream, the trees are not necessarily node-disjoint; as we explain in a later section, node-disjointness is a difficult property to achieve even in Splitstream especially in heterogeneous environments. Note that the true source transmits each packet from each slice exactly once. It does not need to send greater than the stream volume (though it can if it wishes).

Applications can access Chunkyspread through an API that provides *join()*, *quit()*, *send()*, and *receive()* primitives, typical to any multicast protocol. The *quit()* primitive provides functionality for both abrupt and graceful quits, where in the latter case, the member may briefly continue to transmit packets to its neighbors while they find alternates. (Note that the term member refers to receiving

members only, not the true source. We use the terms member and node interchangeably.) Of particular interest is the *join()* primitive that takes the following parameters: the group name, the member type (true source or receiver), the target load, and the maximum load. The two load parameters refer to the transmit load of a member, and may be expressed by the application as absolute throughput values (e.g. 100Kbps), or as a percentage of the stream volume (e.g. 75% or 250%). The maximum load is the absolute maximum volume that the member[3] will transmit at any time while the target load is the volume that the member would like to be sending at steady state. The expectation is that the steady state volume sent by the application will be near the target load: in fact, it may be slightly above or below.

Clearly, it is possible for members to set their maximum loads such that there is not enough capacity in the system to transmit the stream. For instance, if each member sets its maximum load to 50%, no member could receive the complete stream, and the system would fail to operate. It is up to the application to insure that this does not happen. One way to do this might be to "hardwire" the application to always set the maximum load at 150% and target load at 100%. This, of course requires that the member host actually have the capacity to transmit at this rate. Alternatively, the application could measure the capacity of its host, and set the maximum and target loads accordingly. Nevertheless there must be enough capacity overall to transmit all streams to all members. No P2P multicast system can operate otherwise.

While the application (and therefore the application developer or user) operates in terms of a target load, the Chunkyspread protocol itself does not.

[3]Note that the term member refers to receiving members only, not the true source. We use the terms member and node interchangeably.

Chunkyspread internally expresses load in units of the number of slices, and not bandwidth or percentage of stream volume. Chunkyspread uses the following parameters: the number of slices *M*, the latency threshold, minimum node degree *MND*, and minimum load *MinL*. These might be set by the true source and communicated to all members. We will postpone the discussion on the last two parameters to later in this section.

The default value for the number of slices that the stream is split, is 16. We experimented with more and less and this value gave a satisfactory load control as well as an acceptable overhead. The *latency threshold* is a value that determines how the system should weigh the trade-off between achieving target load and minimizing latency. It is expressed as a percentage of the target load. For instance, assume that a given Chunkyspread application requests a target load of 100%, and that $M = 16$ and the latency threshold=10%. 10% above and below 16 slices is 18 and 14 slices respectively after rounding to the nearest slice. The lower edge of the range (14 slices in this case) is called the *Lower Latency Threshold LLT* while the upper edge is called the *Upper Latency Threshold ULT*.

Given the *LLT* and the *ULT*, load balancing and latency reduction work as follows. As long as a given member node's load is outside this range, the system adjusts to move the load within the range. If a node X's load is below its *LLT*, other nodes will try to become a child of X, thus increasing X's load. If X's load is above its *ULT*, existing children of X will try to find other parents, thus decreasing X's load. Once nodes' loads are within the *LLT-ULT* range, they will no longer try to improve load, but rather try to optimize latency. Whenever a change of parent for a given slice improves latency by a certain margin without causing the load to fall outside this range, that change is made.

From this, we can see that a larger *LLT-ULT* range will improve latency at the expense of nodes not getting as close to their target load, while a smaller range has the opposite effect.

To join a Chunkyspread multicast group, nodes must first contact a rendezvous node at a well-known location (DNS name or IP address). This rendezvous node must know of at least one existing member of the multicast group. This style of joining a P2P group is a fairly standard practice, and not further discussed here.

Once a joining member node or the true source finds at least one existing node, it participates in a continuously running distributed algorithm called Swaplinks [66] that produces a random graph among all nodes using simple weighted random walks. This random neighbor graph is the underpinning of Chunkyspread in much the same way as RanSub [79] is the underpinning of Bullet. Swaplinks is able to statistically control the node degree of each node, and Chunkyspread exploits this to give nodes with higher target loads proportionally higher node degrees. The idea here is that nodes with higher load should have more neighbors to transmit slices, and nodes with lower load should have proportionally fewer neighbors. With network churn, the neighbor set of each node changes, but the number of neighbors stays roughly the same. In addition to these random neighbors, nodes may discover other nodes that are nearby with respect to latency. These nodes may be added to the neighbor set to improve latency.

This is where the system-wide parameters *minimum node degree MND*, and *minimum load MinL* come into play. *MND* is the smallest node degree in the random graph that any node may have. Its default value is 8, and as far as

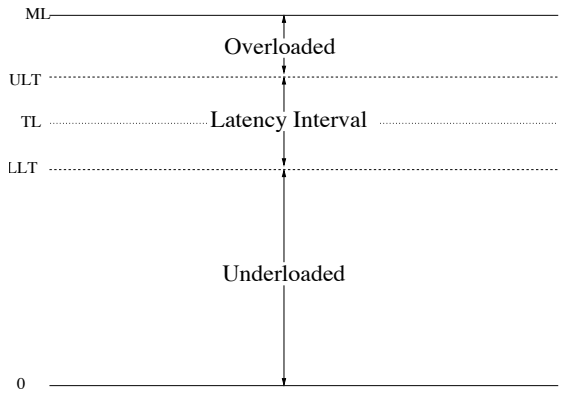


Figure 4.1: The load-latency thresholds

we know, this value is universally appropriate. Since node degree is set proportionally to the target load, the node degree of any nodes is set to be *ND = min[8, (TL/MinL)*MND]*, where *TL* is the target load. The system may choose to allow nodes' target load to be less than *MinL*. In this case, *ND* is set to *MND* (8). As with ensuring that a given Chunkyspread group has enough capacity, the application must also ensure that *MinL* is set to an appropriate value: i.e., the expected smallest capacity of a host in the system. It may also be possible to set *MinL* dynamically, for instance by having nodes remember the lowest *TL* they've seen in the network, and setting *MinL* accordingly. We have not explored this possibility.

Unlike the receiving nodes, the true source discovers exactly *M* (the number of slices) neighbors. The true source transmits one slice to each of these neighbors. These neighbors become the roots of *M* multicast trees, and are called the *slice sources*. If a slice source quits, then the true source discovers this and selects a new random node as the slice source. Note that a node may be a slice source for more than one slice.

A node, upon joining the random graph, tries to find a parent for each slice without forming a loop. We avoid and detect loops using bloom filters in the data packets. In selecting parents, each node tries to maintain a set of constraints, as well as its performance goals and those of its neighbors. The performance goals we have implemented and studied in this work are target and maximum load, and latency, as described above. Other constraints may include tit-for-tat and path-disjointness.

The basic process is straightforward. Each node lets its neighbors know initially about its *LLT-ULT* range and its maximum load (*ML*). Further, each node periodically advertises to all of its neighbors the following: its per-slice bloom filters, information about the arrival time of each slice, its current load (i.e. the number of children it has). Additional performance constraints may be added to this list. Each node takes this information into consideration to determine which neighbors would make appropriate parents for each slice. As conditions change, for example, due to neighborhood alterations, load or latency changes, nodes may select different neighbors as parents for each slice. Note that as a result of this process, a neighbor may be the child for some slices, and the parent for others. Figure 4.1 shows the thresholds used by Chunkyspread in fine tuning the load and latencies in the trees.

Given this overview, the following subsections provide additional detail.

Loop avoidance and detection: Bloom filters offer a spatially efficient method to detect and avoid loops, with a tunable rate of false positives[82]. Each node selects a bloom mask with an appropriate number of bits. A node, before forwarding a data packet, adds its bloom mask to the bloom filter that is tagged along with the data packet. Loops are avoided by having nodes ad-

vertise the bloom filters they receive for every slice to their neighbors. A given node does not select a neighbor as a slice parent if the node itself appears in the neighbor's received bloom filter.

Loops are detected immediately by the first packet that traverses the loop[4]. This packet can either be a data packet sent by the application, or, in the absence of such packets, a probe packet transmitted by a node to its children. The first node to detect the looping packet drops it and immediately selects a new parent.

Fine-tuning Load: As described above, each node periodically checks to see if it has an overloaded parent (above the parent's *ULT*), and an underloaded neighbor (whose load is below *LLT* and satisfies the loop-free condition), and if so attempts to *switch* parents. Since multiple nodes are doing this at the same time, multiple potential switches may be possible. To encourage only the best such switches take place, each node with a potential switch informs its overloaded parent of the loads of all (or a subset of the most) underloaded potential parents. The parent, which may receive similar information from multiple children, picks the best candidate (the child's neighbor with the least load), and instructs the selected child to make the switch. In our system, the overloaded parent usually picks one amongst a set of good candidates so as to avoid implosion of switch requests to such nodes.

The child then sends a switch message to the potential parent which accepts or rejects the request depending on its load and its bloom filter for that slice (these parameters may have changed from the time since the child had made the request). If the switch request is accepted, the child informs the previous parent of the switch completion.

[4]A loop can happen in spite of maintaining a bloom filter. A node that is not yet aware of a bloom filter change in its ancestors, can accept one of the ancestors as its child.

The switch messages that the child sends to its future and the current parent, identify the sequence number of a future data packet at which the current parent should stop transmitting, and the new parent should start. This minimizes packet loss or duplication during the switch itself. The switch message also contains the load parameters that were in force when the decision to switch was made. If these parameters have changed significantly in the interim, the switch is aborted.

It is important to note that, in the absence of churn and switches due to fine-tuning latency, the algorithm for balancing load will converge. Every load balancing switch results in a node above *ULT* reducing its load and a node below *LLT* increasing its load. Once within the *LLT-ULT* range, there are no load-balancing switches that can push a node out of that range, and no load-balancing switches take place between nodes already in the *LLT-ULT* range. The period when the load-balancing switches take place predominantly in a node is called the load-phase of the algorithm.

Fine-tuning Latency: Once all of a node's parents are within their *LLT-ULT* range, the node looks for parent switches that can improve the latency with which it receives packets while keeping loads within the *LLT-ULT*range. This constitutes the latency phase of our algorithm. We use a novel trick that allows us to measure the relative latency with which each neighbor receives each slice without requiring synchronized clocks. Specifically, each node measures the delay at which it receives packets from each slice *relative to other slices*. The idea is simple: a node close to a slice source in a tree will receive packets for that slice relatively *sooner* than it will receive comparable packets of other slices. If a node has a parent that is receiving a given slice *late* (relative to its other

slices), and a potential parent that is receiving the same slice relatively *early*, then it should switch parents (as long as both neighbors' loads remain within range). Note that nodes only make such switches if the expected improvement in latency is beyond a certain threshold. The latency measure described above should be calculated as a moving average to smooth out transient changes due to congestion.

We have not used the overlay path length as a measure for latency reduction for obvious reasons: small path lengths do not necessarily yield low latencies, especially if the underlying graph is locality-aware. A smaller path length does, however, mean that the packet has to traverse fewer nodes which reduces the chances of disconnections in the path. If this is desired, path length can be used as a metric for parent selection (in addition to or instead of latency).

Finally, requesting the best parent (either in terms of latency or load) to supply a slice can lead to an implosion of switch requests at such nodes. This implosion will not just increase the control overhead at such potential parents but will also lead to many of the switches to fail. To prevent this from happening, nodes choose one amongst a set of good potential parents instead of choosing the best parent.

Initial Tree Construction and Forced Parent Selection: In Chunkyspread, new trees must be "kick-started" when the true source first starts the multicast stream or when a slice source quits and the true source chooses a new one. Initial tree construction involves a simple controlled flooding mechanism similar to the one used in Chainsaw. Shortly after a node starts receiving flooded packets for a given slice, it selects a parent from among the neighbors from which it received the flooded packet. The selected parent may reject the request if not doing so

would push its load above its maximum load *ML*, but otherwise must accept the child.

Apart from this, a node that joins a multicast session whose trees have already been constructed through the flooding mechanism described above, may have to periodically request its neighbors to be parents for each of its slices until it finds them. As a result of these cases, the parent's load may exceed the upper latency threshold *ULT*. Normally, the ongoing load balancing process will bring the load back to or below *ULT*, though on the rare occasion a node's load may stay above *ULT* for a period of time due to the lack of availability of potential parents for its children (though there may be underloaded nodes elsewhere in the system).

There are three other cases where a node may request a parent even though doing so pushes the parent's load above its *ULT*. All three are cases where the node is forced to change its parent. This may happen when a loop is detected, when the parent quits the group, and when the Swaplinks algorithm changes the neighbor set as part of its normal operation[66]. While the first two is effectively a temporary disconnection from the tree, the third is usually similar in effect of any normal switch. Note that a node may only reject a request to become a parent if doing so pushes its load above *ML*, or it does not satisfy the looping constraint (and any other if needed).

4.3 Overview of Splitstream

Since we make simulation comparisons of Chunkyspread and Splitstream, a brief overview of Splitstream is provided here. Splitstream builds multiple trees

on top of Scribe, a single-tree multicast protocol that constructs its tree using the overlay routes of the underlying DHT (Pastry). However, a node may not have enough capacity to serve all its in-neighbors that want to join the multicast group. In order to avoid nodes getting loaded beyond their capacities, Scribe resorts to two other mechanisms, namely pushdown and anycast operations. When a fully loaded Splitstream node is requested to parent another node, it may preempt one child node for another based on ID constraints [78]. The resulting orphaned node recursively contacts the parent's descendants (called *pushdown*) to find a parent and if it still cannot find one, *anycasts* to the group of nodes that have spare capacity.

When a fully loaded node C gets a request from a potential child A, it can choose to either drop one of its current children B based on whether A overlaps with C's ID more than B. The orphaned or preempted node (A or B) then contacts one of C's children and the process continues recursively. This is called the pushdown operation. If there still remains an orphaned node after the pushdown operation, it contacts a group maintained by Scribe comprising of nodes that have excess capacity left. A depth-first traversal is made on this group to find a node that can provide the stream to the orphaned node[71]. This is the anycast operation.

Splitstream works well in homogeneous cases with usually the Pastry neighbors serving the nodes. However, in heterogeneous environments, the pushdown and anycast operations happen more often and this leads to frequent disconnections of nodes: not only is the preempted node disconnected, but so are its descendants in the tree. The two operations lead to the formation of parent-child links that are apart from the underlying Pastry neighbors. Hence, Split-

stream starts losing the benefits of cycle-free and route-convergence guarantees offered by the underlying DHT as the number of non-Pastry neighbors increases. In short, Splitstream prefers ID-based constraints over load constraints when initially creating the tree and this leads to further complications in the tree-building protocol.

4.4 Results

We have performed a series of experiments on a packet-level, event-driven simulator coded in C++. We have also implemented the system and made some simple deployment experiments on Emulab. The default number of member nodes in each simulation is 5000. The Chunkyspread simulation could operate with more nodes and higher join rates than the ones specified in this section, but the Splitstream simulator could not, so we limit our simulations to 5000 members. To calculate the latencies between members, we placed member nodes at random edge locations on GT-ITM network topologies having 5050 routers [75], and set delays proportional to the distance metric of the resulting topology. We chose to select a very pessimistic value for network latencies: the median latency is around 400ms, and the maximum is roughly 650ms. As a result, the convergence times shown are worse than one might expect over the commercial Internet (for both Chunkyspread and Splitstream). We assume that control messages are sent over TCP, and so ignore message loss in our simulations.

The random overlay is constructed using a packet-level trace file generated offline by a Swaplinks simulator. The trace file allows us to determine the delays associated with the neighbor selection in Swaplinks. The trace file was used in

order to avoid running the random neighbor selection as part of the simulator, hence making the simulations faster. To further scale the simulations, the simulator does not explicitly generate data packets. We do, however, calculate the amount of time it would have taken for a packet to travel node to node. This calculation is needed both in determining when bloom filter information arrives at each node, and for calculating the relative slice arrival time used to improve latency.

Member nodes in the simulation receive all slices. In principle, it would be possible for nodes to receive some fraction of the slices and still be able to reproduce the stream, for instance, by using Multiple Description Codes[85]. We neither implemented nor simulated this. The default number of slices in our simulations is $M = 16$. To model heterogeneity, each node is assigned a uniformly random node degree within a specified range. By default, the range is from 8 to 50 inclusive, thus producing a roughly 6x range of loads. This represents a moderate level of heterogeneity, representing say a population of users behind dial-up modems and broadband, or behind broadband and T1. The target load TL for each node is derived from its node degree in such a way that the sum of target loads across all nodes is approximately equal to the total volume needed to transmit the stream to all nodes. This results in default values for TL being distributed uniformly between 4 and 28 slices (the median of 4 and 28 is 16, the number of slices). The default setting for maximum load is $ML = (1.5)TL$. In other words, the total capacity of the system is 50% more than what is needed to transmit the stream to all nodes. This represents a well-provisioned system: something required in any event to get good performance[76].

We experiment with two settings for the *LLT-ULT* range. One is when there

is no latency range (*i.e., ULT=LLT=TL*), resulting in no latency optimizations whatsoever. This is denoted *Lat0*. In the other, they are set to *2(TL)/16* slices from *TL* (rounded up for *ULT*, and down for *LLT*). In other words, if *TL=16*, then *LLT=14* and *ULT=18*. This is denoted *Lat2*.

We chose a bloom filter size of 128 bits and a bloom mask size of 6. This yields a false positive rate of 0.25% after insertion of 10 keys. The heartbeat period is set to 1 second and the timeout period to detect a node failure is set to 4 seconds. Parent switching decisions are made every second.

We compared Chunkyspread simulations with those of Splitstream, which uses a simulator coded in C# that was provided to us by Miguel Castro. Wherever possible, we provide apples-to-apples comparisons of Chunkyspread and Splitstream. For instance, the Splitstream simulations are run over the same GT-ITM synthetic routing topology and have 16 slices. Splitstream does not, however, have parameters analogous to target load *TL* and upper and lower thresholds *ULT* and *LLT*. Rather, Splitstream provides a single parameter, maximum load (*SML*). *SML* is analogous to Chunkyspreads *ML* in that the load never exceeds *SML*. It is unlike Chunkyspreads *ML*, however, in that a Splitstream node may easily settle on a sustained transmission rate of *SML*, whereas an Chunkyspread node may temporarily transmit at *ML*, but will quickly move towards the *LLT –ULT* range. As a result, we need to interpret *SML* differently from *ML*, and an apples-to-apples comparison is not really possible. Specifically, *SML* means a transmission rate at which I would be perfectly happy to operate, whereas *ML* means a transmission rate that I am capable of achieving for brief periods, but would rather not.

Because of this difference, in one case we treat *SML* to be equivalent to

ML (denoted *SS(1.5)*). That is, we set it to be 50% above the number of slices (*SML=1.5TL*) where *TL* is the target loads for the corresponding nodes in Chunkyspread. In the other case, however, we try to treat *SML* as though it were equivalent to *ULT*. As such, we set *SML=(1.2)TL* to compare with *Lat2* (denoted *SS(1.2)*). To compare with *Lat0*, we tried setting *SML=TL*, but Splitstream does not converge in this case, so instead we use *SML=(1.1)TL*, denoted *SS(1.1)*. Splitstream has a time-out parameter that determines how long a node should wait for the result of an anycast operation before trying again. This parameter is set to 4 seconds. A value less than this tended to result in too many unnecessary anycast operations.

We have broadly considered four scenarios to evaluate our protocol.

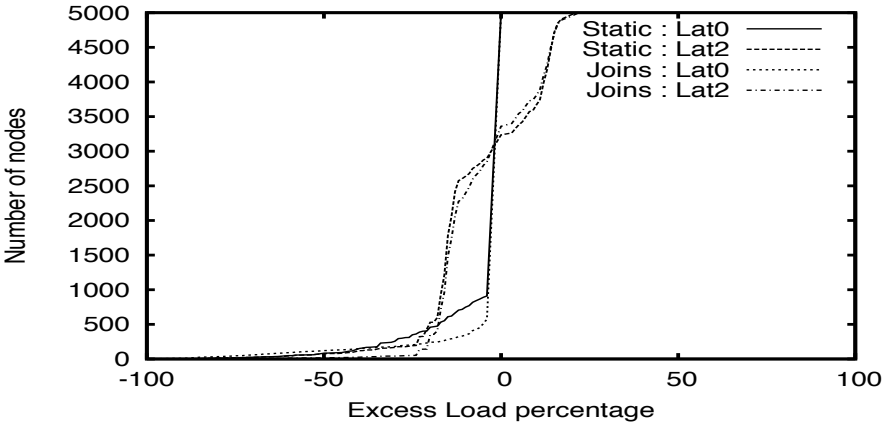
Static scenario: This corresponds to the case when all overlay nodes are present in the network right from the beginning of the simulation. This means that the random graph is constructed completely, even before the *true source* starts building trees to kick-start the multicast session. This scenario is not a very realistic one but is useful in analyzing just the performance of our load-latency algorithm without the influence of any churn. The Swaplinks simulator did not have functionality provided for locality-awareness. To determine the effect of adding locality to the random graph, we have run static simulations where, in addition to the random neighbors selected by Swaplinks, some number of nearest neighbors were added to the neighbor set of each node. For all the other scenarios which involve churn, we did not incorporate locality since the nearest neighborhood set alters with churn, and we did not want these changes to affect the degree invariant guarantees offered by Swaplinks.

Join scenario: There are 3750 overlay nodes in the network (similar to the static case) and the rest (1250 nodes) join at a rate of 50 joins per second from the 20^{th} second by which time most of the originally present nodes had reached a steady state. This scenario depicts a more realistic picture than the previous one; it can possibly be a live event that attracts a large audience within a short span of time.

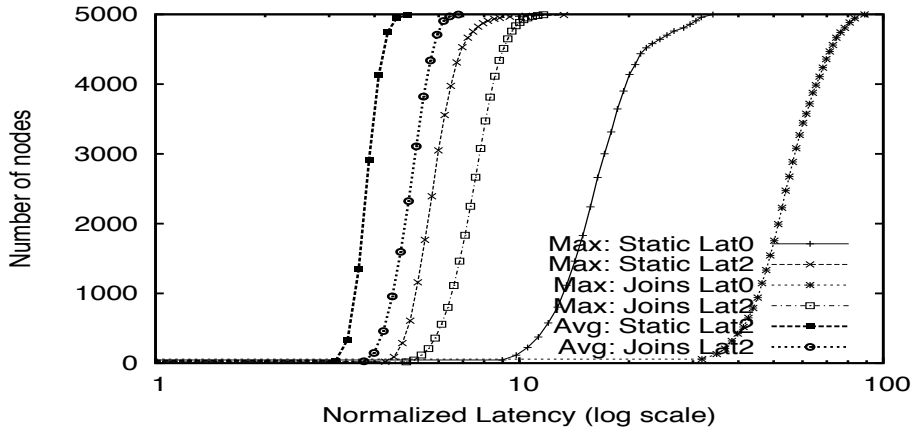
Bursty failures: There are 5000 nodes in the network and a percentage of the nodes fail at the *same time instant*. We consider two cases: One when 10% of the nodes fail (small burst) and the other when 50% of the nodes fail (large burst). These pathological cases may not be very close to realistic scenarios, but do help in analyzing the robustness of the protocol against node failures. Such a high failure rate could potentially lead to network partitions, but Swaplinks was resilient enough to prevent them from happening.

Churn scenario: To understand the effect of more realistic scenarios on our protocol, we simulated Chunkyspread under continuous churn in which nodes join and leave at the same time. The scenario that we have studied is similar to the one tested in [78]. We consider Poisson arrivals at 10 joins per second, and pareto stay times with a minimum duration of 90 seconds and a mean of 300 seconds (which implies the pareto parameter $\alpha = \frac{10}{7}$). Pareto is a heavy-tailed distribution which is typical of the behavior of users in such environments[76]. The churn happens for the first 1000 seconds after which the remaining live nodes are allowed to settle down for the next 200 seconds.

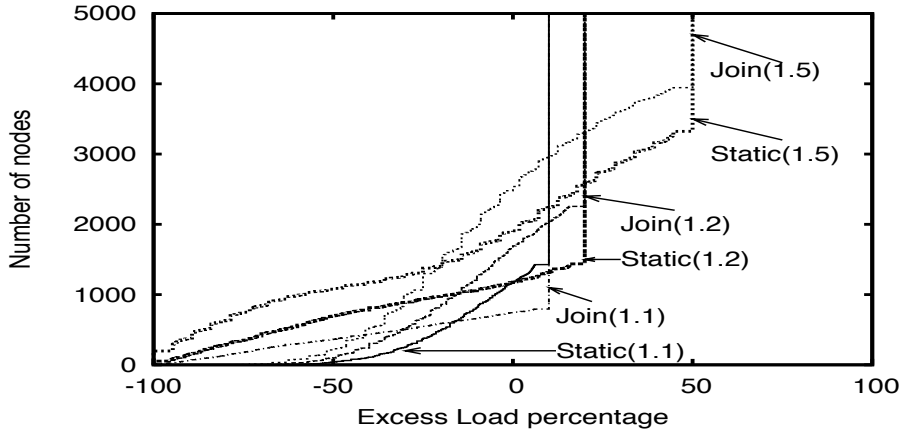
The static and the join scenarios We first present a comparison study be-



(a) Load distribution in Chunkyspread

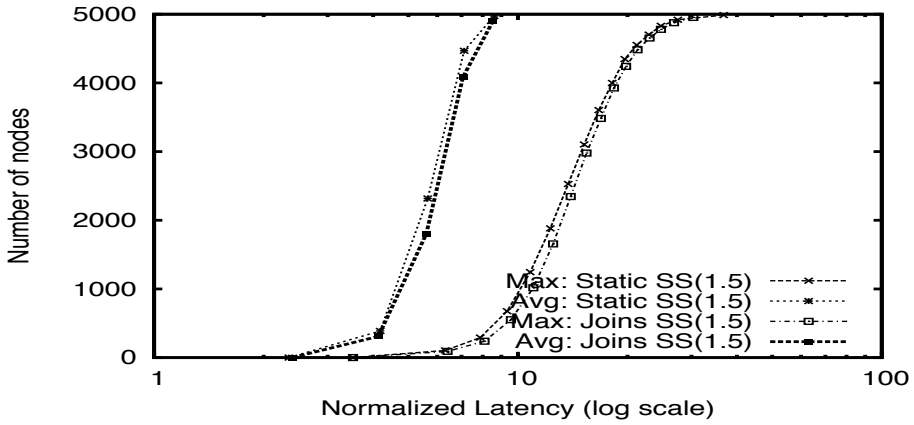


(b) Maximum and average latency distribution in Chunkyspread

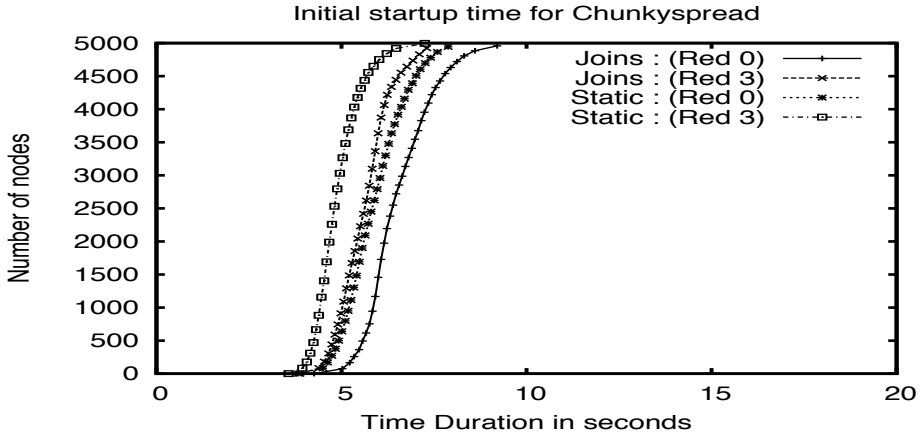


(c) Load distribution in Splitstream

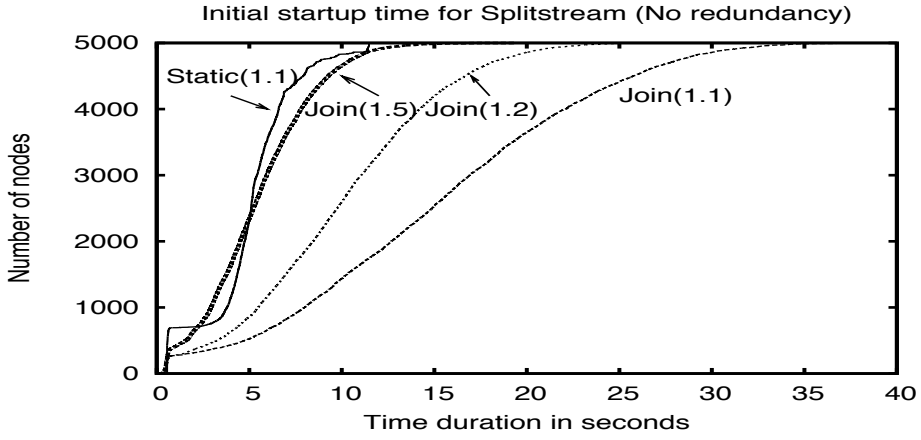
Figure 4.2: Load and Latency experiments



(a) Maximum and average latency distribution in Splitstream



(b) Initial Startup time in Chunkyspread



(c) Initial startup time in Splitstream

Figure 4.3: Startup times and Latency

tween Splitstream and Chunkyspread followed by an evaluation on the convergence and the control overhead of Chunkyspread.

Comparisons with Splitstream

In the first set of experiments, we analyze the tradeoff between load balance and latency in Chunkyspread and compare them with Splitstream. We introduce the term *excess load percentage* to quantify load in the protocols. It is defined for every node as follows.

$$Excess\ Load\ Percentage\ =\ \frac{Node's\ Load\ -\ TL}{TL}\% \tag{4.1}$$

This parameter quantifies how close nodes reach their target load and hence the degree of fairness provided by the protocol. A value of 0% implies that the node has perfectly reached its TL, while a value of -100% means that the node has zero load. The maximum value of this parameter is bound by $\frac{100.(ML-TL)}{TL}\%$ which is 50% in our Chunkyspread simulations.

We use two parameters to evaluate the latencies: the maximum and the average overlay latencies over the slices obtained at each node. The latencies are normalized with respect to the median value of the network latencies between overlay nodes. We chose not to use the *network stretch* parameter to evaluate our latencies. Network Stretch is a common term used in the literature that is defined as the ratio of the measured overlay latency to the network latency between the true source and the node. Network stretch may not give a true picture of what the latencies are: for example, a high network stretch could actually be due to high latency or could be due to a low network latency with the true source.

Figure 4.2(a) shows the cumulative distribution function (cdf) of the excess load percentage of nodes in Chunkyspread after steady state was reached. We observe that *Lat0* performs quite well in both the static and the join scenarios: more than 80% of the nodes reach exactly their *TL* in the static scenario while around 90% of the nodes reach their *TL* in the join scenario. With the latency phase added, Chunkyspread still performs well: almost 90% of the nodes are within 25% of their *TL* values in the *Lat2* case in both the join and static scenarios. The maximum fraction of excess load that any node reaches is about 20%. Apart from the good load balance, we observe comparable performances of the join and the static cases which indicates that the protocol can function at high join rates as good as in cases without any churn at all. The heavy tails observed on the negative side of the x axis in these curves are because of imperfect configurations of node connectivity.

Figure 4.2(b) shows the cdf of the maximum and average overlay latencies normalized with the median of the network latencies between nodes in the network. The x-axis is shown in log scale. The cdfs have been plotted for the *Lat0* and the *Lat2* cases. We first observe that *Lat0* yields very high latencies in both the static and the join scenarios, which is expected since *Lat0* is completely 'latency-blind'; this can be seen from the maximum latencies of *Lat0* in both the static and the join cases. We observe significant improvements in latencies with *Lat2*. The 90^{th} percentile values of the maximum latencies in both the static and the join cases are around 7 and 9 respectively while the same for the average latencies are around 4 and 6 respectively. The difference between the maximum and the minimum latency values gives us an idea of how long it takes to receive all the slices for the same block of the stream and hence the size of the application buffer required to counter losses while waiting for the full block. We note

that the latency for any slice experienced by a node is bounded below by its network latency to the true source. Then, for example, if we assume the median network latency were around 50 milliseconds, then a 500 millisecond buffer is necessary to successfully play out the stream *in the steady state* even if losses due to factors such as churn or congestion are not considered.

The overlay stretch of a node is defined as the ratio of the average latency observed over its slices to its minimum latency in the overlay graph with the true source. It is a measure of the quality of the latency algorithm just with respect to the underlying random graph constructed. We see that over 90% of the nodes have overlay stretches between 1 and 2 for Lat2 in the static scenario; the join scenario also incurs similar values. This shows the good performance of the latency phase of our algorithm.

Let us now see how Splitstream fares with respect to load and latency. Figure 4.2(c) shows the cdfs of the excess load percentage values for SS(1.1), SS(1.2) and SS(1.5) for each of the join and static cases. As expected, a considerable number of nodes get saturated to their *SML* values and the percentage of such saturated nodes increases as $\frac{SML}{TL}$ values decrease. For example, the percentage is 35% for SS(1.5), 60% for SS(1.2) and 85% for SS(1.1) in the join cases. This is in stark contrast to the excess load percentage distribution that Chunkyspread's *Lat2* and *Lat0* yielded. We also find that the join case has a worse load balance than the static case, since the newly joined nodes are not provided enough opportunities to supply the slice unless an orphaned node or another newly joined node requests for a slice. In Chunkyspread, the load balance algorithm ensures the newly joined nodes also participate in supplying the slices.

The graph in Figure 4.3(a) shows cdfs of the average and maximum latencies

in the static and the join cases. We note that both the average and the maximum latencies showed very marginal improvements as $\frac{SML}{TL}$ was increased with both the static and the join scenarios performing comparably. The comparable performances show that curbing the spare capacities do not have a significant effect on the latencies. We have presented only SS(1.5) here for clarity. The 90^{th} percentile values of the average latencies for both the static and the join scenarios are close to 8; this is greater than Chunkyspread's *Lat*2 values but still quite comparable. However, the maximum latencies show really high values. SS(1.5) yields 90^{th} percentile values of around 20 in both the static and the join scenarios; it also displays a heavy tail, almost reaching 30. These are in fact comparable with (static) Chunkyspread's *Lat0* values. The reason for the high maximum latencies is that with heterogeneity, more (random) non-DHT parent-child links are formed which are not necessarily latency-optimized unlike their DHT counterparts. The huge difference between the average and the maximum latencies requires an application buffer of considerable size and this buffer is to just counter losses due to delays in the slice arrivals for the same stream. In the example that we had considered for Chunkyspread above, Splitstream nodes may require a 1.5-second buffer in the steady state just to counter losses due to late arrival of slices.

We observe a similar trend with the cdfs of maximum hop length (from the true source) across all slices received by each overlay node as shown in Figure 4.4. As we had already mentioned, higher hop lengths relate to a lower tree resilience, since nodes are more prone to disruptions from the trees due to the failure of one or more ancestors in the path to the true source. From the graph, we see that Lat0 yields very high hop lengths. We also see that there is a good improvement with Lat2 which yields 90th percentile hop lengths of around 8 in

both the static and the join scenarios. We note that the static scenario performs comparably with the join scenario, though it had outperformed the latter in the latency figures, as we had discussed above. This is because the static case builds locality-aware graphs which usually yield lower latencies at the cost of greater hop lengths. Again, Splitstream performs much worse than Chunkyspreads Lat2: with 90th percentile values as high as 30. The reason why this happens has been discussed above.

We define the *initial startup time* of a node as the time taken since its joining the multicast session, for it to start receiving the entire stream. In Chunkyspread, a newly joined node initially gets its stream by periodically pinging its neighbors (as described in Section 4.2) and this mechanism is independent of the choice of load-latency parameters used. While this quantity is clearly defined in Chunkyspread, it is not in Splitstream, since a node that has started to receive its stream from all its trees can potentially get orphaned from one or more trees. Hence, we include all the time durations during which nodes are disconnected from the tree due to such preemptions, into the initial startup time. Note that the disconnection due to orphaning a node will lead to disconnections of its descendants in that tree, if any.

Figure 4.3(b) shows the cdf of the initial startup time for Chunkyspread. We find that the 90^{th} percentile value in the join scenario is about 8 seconds while it is 7 seconds in the static scenario. The reason for the difference is the fact that the static scenario is run with locality which enables faster tree construction. In the graph, *Red 3* denotes the case where the stream is encoded with 3 redundant slices, hence it is enough if the node gets any 13 out of the 16 slices to obtain the full stream. We find that in the static case, the 90^{th} percentile value for *Red 3* is

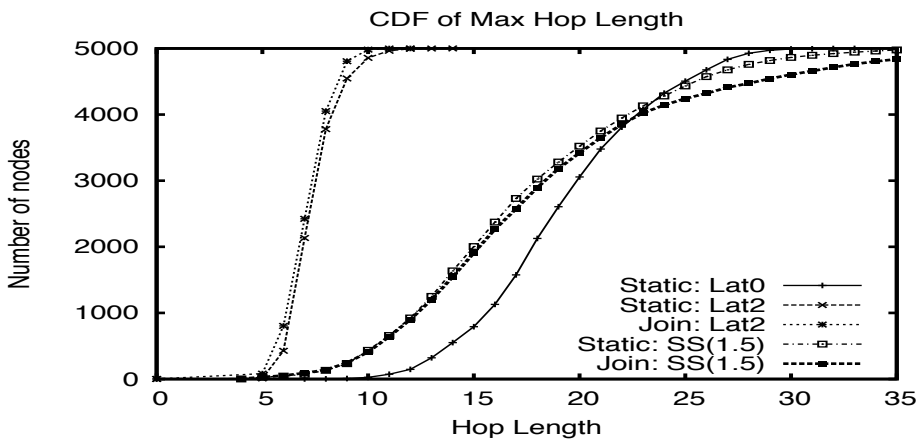
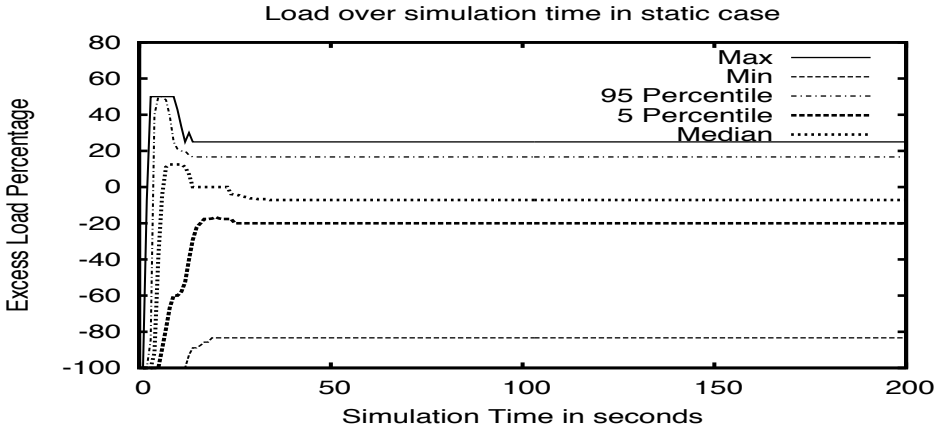


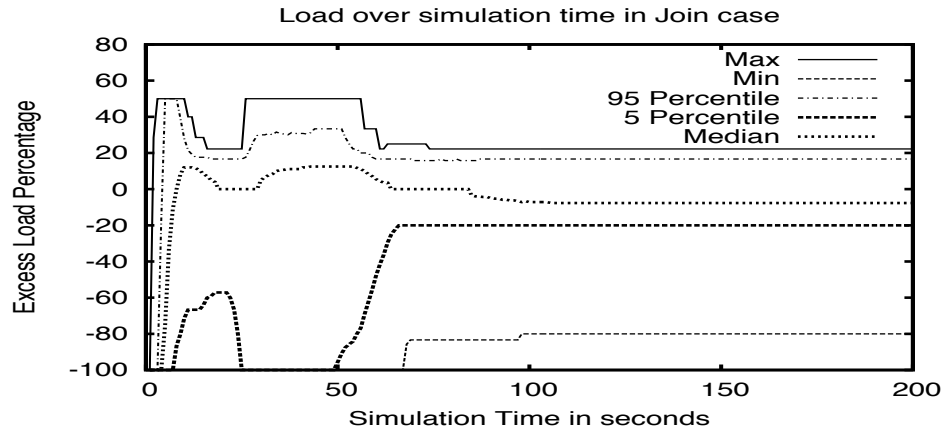
Figure 4.4: Hop Length Distribution

less than 6 seconds.

Figure 4.3(c) shows the initial startup times of Splitstream. As claimed in [72], the system performs well in the static case with even SS(1.1) yielding a 90^{th} percentile value of around 8 seconds which is comparable with Chunkyspread's values. We see that as the spare capacities in the system decrease, the initial startup time increases as seen by the curves for the join scenario. This is expected, since, with lesser spare capacity, more time has to be spent during the pushdown and the anycast operations. SS(1.5) performs comparable to Chunkyspread in the join scenario, with a 90^{th} percentile value of around 9 seconds. But with decreasing $\frac{ML}{TL}$ values, the startup time shoots up to 17 and 26 seconds for SS(1.2) and SS(1.1) respectively; this is in a good contrast to the static case. As nodes join, many of the existing nodes have already been saturated to their *SML* values and the newly joined nodes result in more anycast and pushdown operations. We note that with Chunkyspread, the load balance algorithm ensures that the spare capacities are distributed across nodes even when nodes are joining at a high rate.



(a) Load of nodes over the simulation time in static case (Lat2)

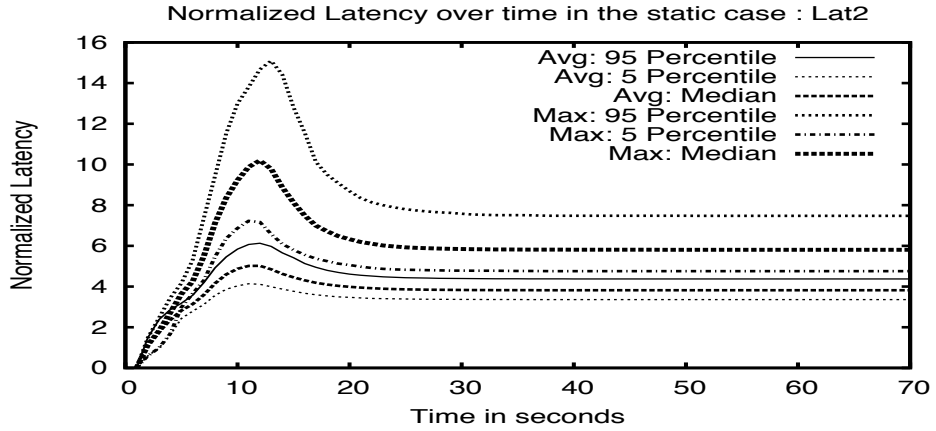


(b) Load of nodes over the simulation time in join case (Lat2)

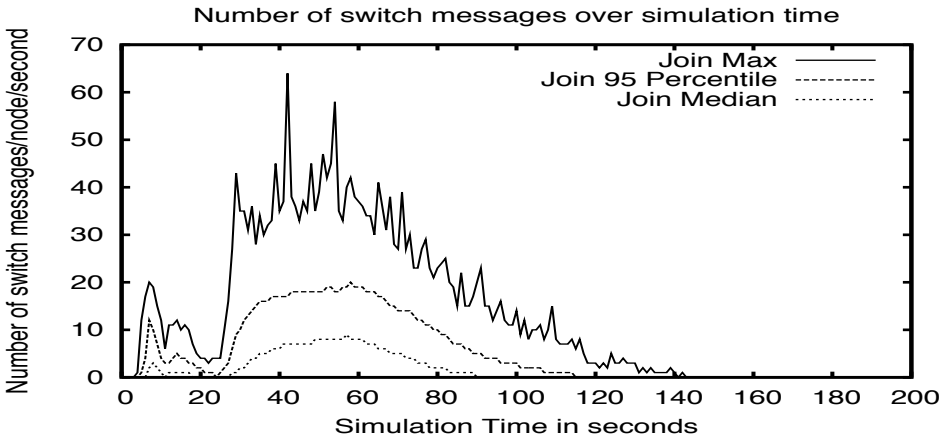
Figure 4.5: Load across simulation time

Time to convergence

We now assess the convergence properties of our algorithm. For our protocol, the convergence time is the time taken till the last switch is successfully completed. We noted for every node the last time instant that it had completed a switch in the system. We observed that *Lat0* converged quite well in both the static (18 seconds) and the join (70 seconds) scenarios. We also saw that while *Lat2* converged within 60 seconds in the static scenario, it took around 120 sec-



(a) Maximum and average latencies over the simulation time in the static case



(b) Control Overhead in Chunkyspread: Switch messages over simulation time

Figure 4.6: Latencies and control overhead over simulation time.

onds to converge in the join scenario, which was 75 seconds after the last join took place. In contrast, Splitstream reaches a steady state as soon as the last orphan node gets a parent. Hence its convergence time is actually the startup time that we discussed earlier.

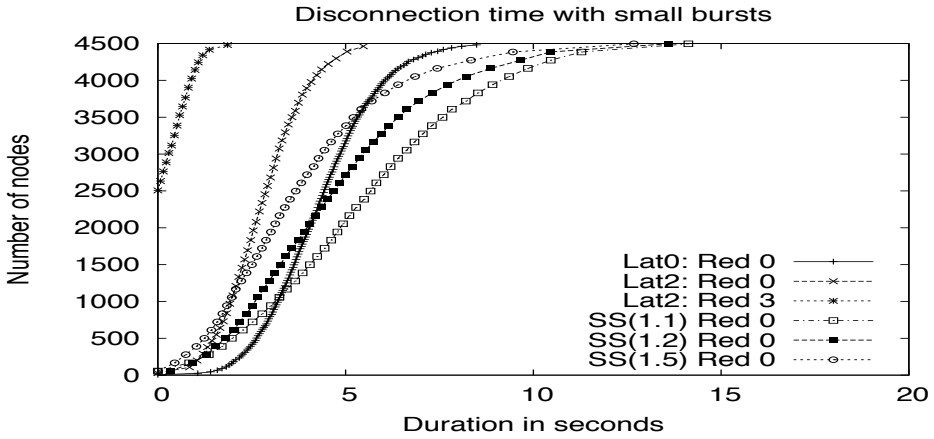
Figures 4.5(a) and 4.5(b) show the excess load percentage per node as the simulation proceeds in the case of *Lat2* for the static and the join scenarios respectively. The maximum and the 95^{th} percentile curves in the static case peak

to ML during the first 10 seconds of the simulation after which the load phase of the algorithm brings both the curves down to within the target upload interval. This shows that our algorithm can distribute the loads fairly across nodes quite fast, so that if more nodes were to join, there is a good chance that there is spare capacity within their neighborhoods. This is in fact depicted in Figure 4.5(b). In the join scenario, we can observe a peak during the first ten seconds; the second peak arises after nodes start joining and stays till 10 seconds after the last node had joined the network. Though there are nodes saturated to their ML values (50%) during this time period, the 95^{th} percentile curve stays roughly at 30% while the median hovers around 10% during this time which show that there are a considerable number of nodes with spare capacity that can serve a newly joined node quite fast.

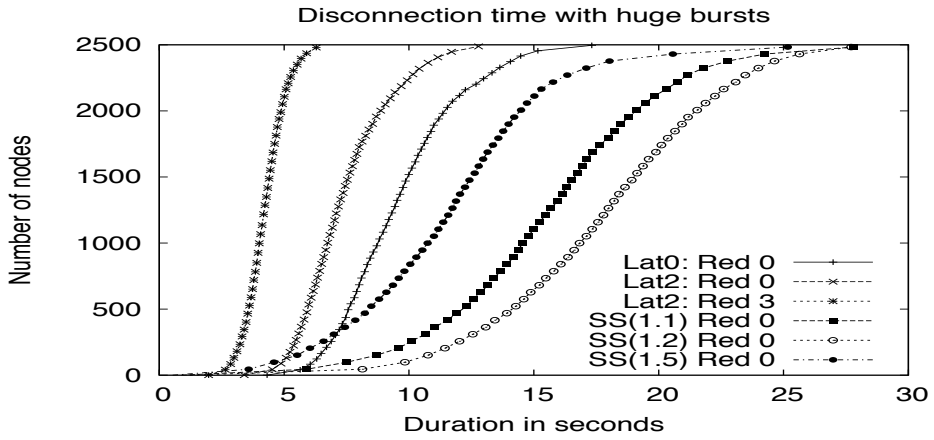
Figure 4.6(a) shows the normalized average latency over the slices of nodes as the simulation proceeds in the static scenario. We observe that the load phase of the algorithm shoots the latency up initially, but then, the latency phase of the algorithm steadily brings it down. The peaks in the 95^{th} percentile curves of the average and the maximum latency values show that Chunkyspread may need to maintain an application buffer of a considerable size for the temporary period of time when the load phase of the algorithm is more dominant than the latency phase; such cases happen after there is churn or after the true source kickstarts the multicast session.

Control Overhead

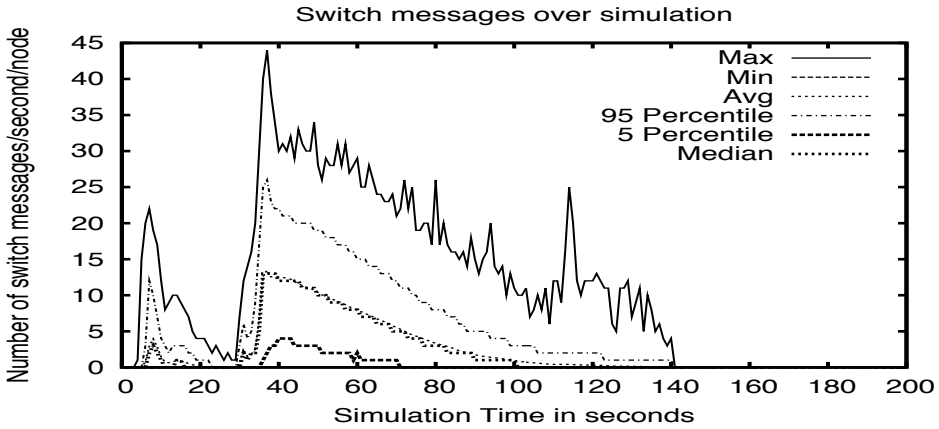
Next, we evaluate the the control overhead incurred by nodes in the network due to switch messages. Figure 4.6(b) shows the number of switch messages



(a) Recovery duration for 10% burst

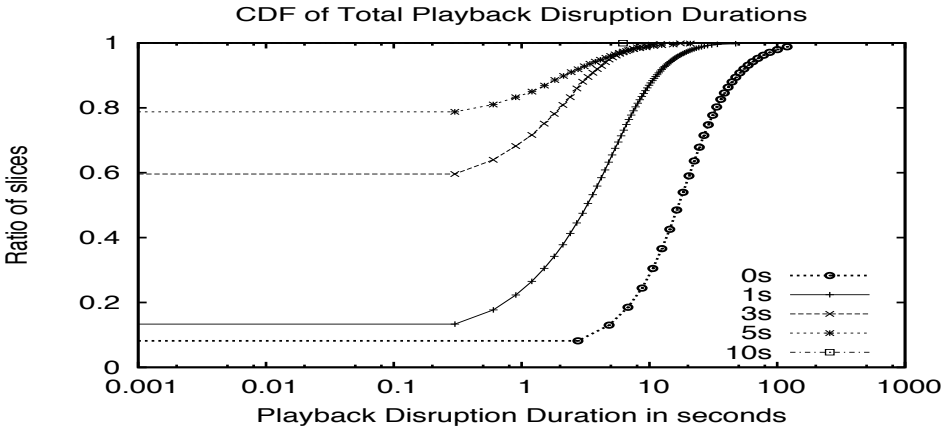


(b) Recovery duration for 50% burst

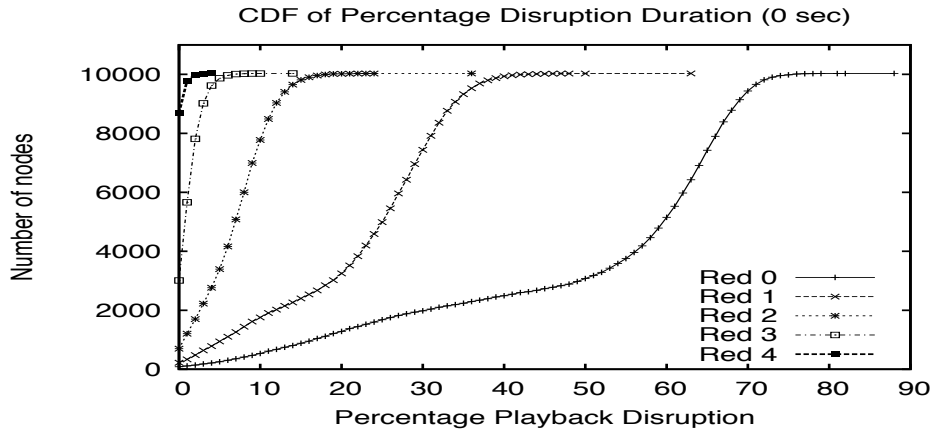


(c) Control Overhead with 0s buffer

Figure 4.7: Response to failures



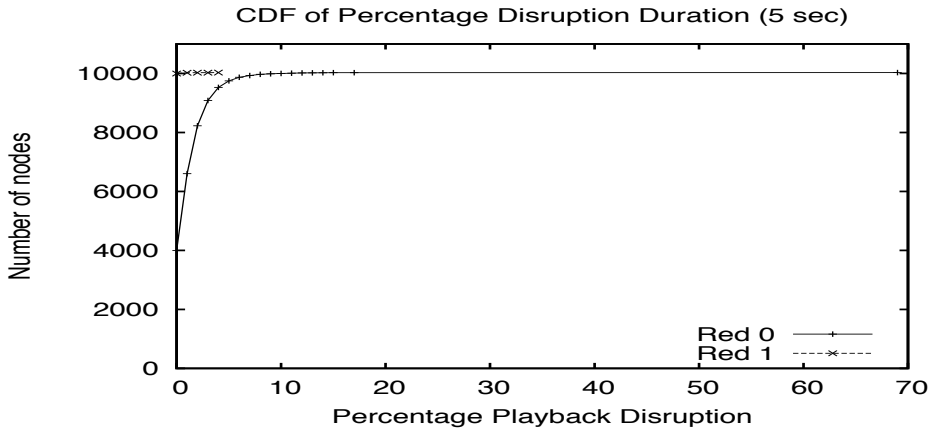
(a) Total playback disruption duration across slices for various buffer sizes



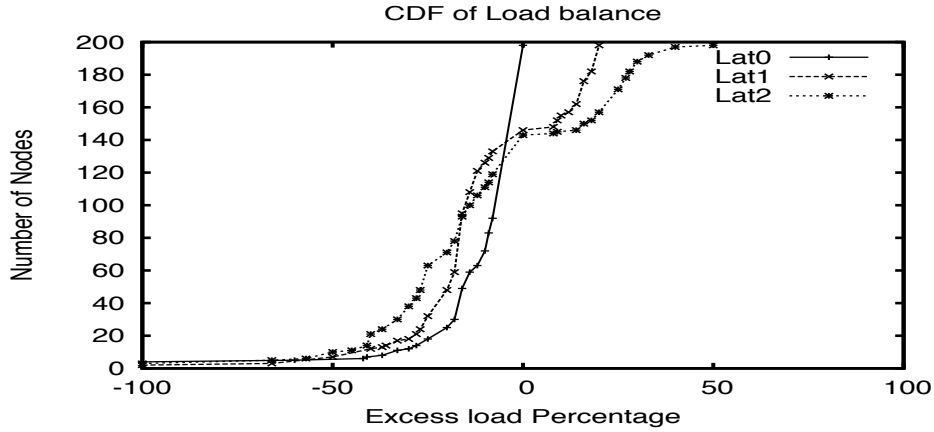
(b) Percentage playback disruption duration with 0s buffer

Figure 4.8: 0s buffer Experiments

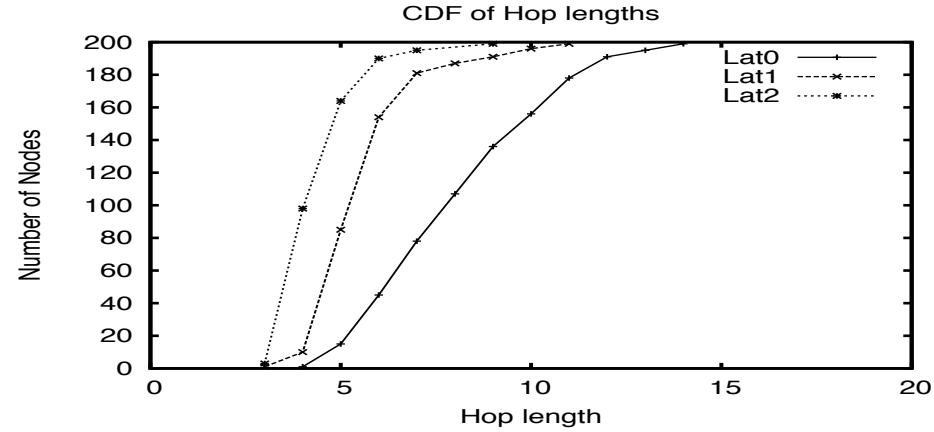
sent per node per second over the simulation time of 200 seconds when *Lat2* is run in the join scenario. The peaks correspond to the time when nodes are joining the system and also after the true source kickstarts the multicast session. Though the dominant peak value of the maximum number of switch messages sent by any node is 60, the peak values of the 95^{th} percentile and the median values of the switch messages are about 20 and 8 messages per second per node respectively. This indicates a modest overhead amongst Chunkyspread nodes even at a high join rate. Apart from this, we observed that around 50%



(a) Percentage playback disruption duration with 5s buffer



(b) Emulation result: CDF of Load Distribution



(c) Emulation result: CDF of Hop lengths

Figure 4.9: Playback disruption and Emulation results

of the switch messages sent during the joining phase account for failed switch requests.

We observe that a switch is usually a three-party negotiation but is asymmetric in the number of switch messages sent at each node: A load-based switch originates from the original parent and ends at it; this involves a total of 4 messages. A latency-based switch involves 5 messages since it starts from the child seeking the current parent to let it switch. Other kinds of parent selections (the forced ones) involve two or three switch messages depending on whether the old parent is also involved in the switch or not.

There can be switches which are rejected either at the child or at the potential parent. Hence the number of switch messages may not exactly correspond to the number of successful switches. We examined the number of successful switches at each node over the course of the simulation time. We observed that in the join scenario, the dominant peak value of the maximum number of switches is around 12 per second per node and happens during the time nodes join, while the 95^{th} percentile value is around 3 switches. We found during this period that almost 50% of the switch requests fail due to conditions that we had already discussed.

Bursty failures: To quantify data losses due to node failures, we measure the time during which nodes are disconnected from one or more trees. We measure the *recovery duration* for each node, which is defined as the time duration calculated from the instant nodes *detect* failures of their neighbors till they get connected back to the trees. It is to be noted that during the recovery period, nodes are disconnected from the tree and so are its descendants. Hence, while a node is trying to recover from a parent's failure, this duration that its descen-

dants are disconnected also get accounted to the descendants' recovery duration (since an ancestor is trying to recover on their behalf). We note that, with no redundancy in the 16 slices, if a node is disconnected from even one slice tree it gets accounted in the recovery duration.

Figure 4.7(a) shows the cdf of the recovery duration when 10% of the 5000 nodes fail at the 30^{th} instant, at various levels of slice redundancy. We find that both the protocols recover quite fast with 90^{th} percentile values of about 5 seconds and 8 seconds in *Lat2*, and SS(1.2) respectively. *Lat2* performs better than *Lat0* primarily because the former yields lesser hop lengths which, as mentioned before, leads to better resilience. On adding redundant slices, we find a drastic improvement in the recovery times. For example, with a redundancy of 3 slices, more than 50% of the Chunkyspread nodes are not disconnected at all and the maximum recovery duration is around 2.5 seconds. We also find that Splitstream nodes are disconnected for longer durations as $\frac{ML}{TL}$ values decrease.

Splitstream performs worse than Chunkyspread when 50% of the nodes fail at the same instant. Figure 4.7(b) shows the recovery duration in such a scenario. With *Lat2*, the 90^{th} percentile recovery time is 10 seconds while it is at least 15 seconds for Splitstream. When redundancy is added, there is a good improvement in the recovery duration: the 95^{th} percentile value for Chunkyspread is just 5 seconds in the case when 3 redundant slices are added. This just goes to show Splitstream's inability to handle a huge failure burst. The problem, we suspect, is the high hop lengths that Splitstream incur (as we had seen in 4.4), which affects its robustness to node failures.

Figure 4.7(c) shows the number of switch messages over the course of the simulation in the 10% burst case with *Lat*2. We see that the maximum control

overhead peaks at 42 messages per second per node just after failures are detected while the median value peaks to 12 messages per second per node. This represents a moderate amount of overhead.

Effect of other parameters: We tried to see the effect of altering parameters such as the number of slices, degree of heterogeneity and the number of neighbors on the protocol. We largely observed that these parametric changes do not result in significant changes to the protocol performance.

We studied the performance of our protocol on varying the number of slices that the stream is split. We found that on increasing the number of slices from 8 to 32, there was an improvement in the load balance: in the static case of *Lat*0, around 90% of the nodes reached their TL values when multicast with 32 slices while 90% of the nodes were within 20% of their TL values when multicast with 8 slices. The more striking feature between the two cases was the tail; the least loaded node had an excess load percentage of -25% for 32 slices and -60% for 8 slices. The better performance with increased slices can be explained by the fact that more slices offer a finer granularity in controlling load. But this is at the cost of proportionally more number of switch messages, hence more control overhead. Increasing the number of slices does not have any bearing on the latencies though.

Another set of experiments was conducted to study how neighborhood sizes affect our algorithms. To simplify our study, we considered a static homogeneous scenario with all nodes having a target upload of 16 slices.We tested our protocol with graphs having constant degrees of 12, 24 and 36 for the Lat2 case. We observed that as the number of neighbors was increased, more nodes were closer to their target loads, though the improvement was not significant. In the

10% burst case, we found that when the number of neighbors was increased from 12 to 24, the recovery duration improved the 90^{th} percentile values from 4 seconds to just 2 seconds. This is because with more neighbors, a node disconnected from the tree has a greater chance of picking up a parent from its neighbors that can supply the slice.

We also experimented with varying levels of heterogeneity in the network. We tested the simulator with three scenarios: (a) the homogeneous scenario (in which number of neighbors is assigned to 32 for all nodes), (b) moderate heterogeneity (similar to our default case), and (c) high heterogeneity (in which the number of neighbors is distributed between 8 and 200 neighbors). As expected, we observed that (a) did better than (b) and (c) with respect to load balance, but the improvement was not significant. In scenario (a) all the nodes were within 12.5% (or $\frac{2}{16}^{th}$) of the TL value which is 16 for all the nodes. Scenario (b) is the curve obtained in 2(a) for the static $Lat2$ case. Scenario (c) performed very close to (b); from these numbers, we can infer that our protocol performance is independent of the degree of heterogeneity in the system.

Churn scenario: We have so far considered isolated node joins and failures in our simulations. As we had already noted, a more realistic churn scenario would be to consider one in which nodes join and leave at the same time. We had already discussed in the beginning of this section, the join and stay time parameters used in simulating the churn.

The disconnection time intervals are noted at every node for every slice; these are the time intervals when the node is disconnected from the slice tree due to an ancestor's failure. After obtaining the disconnection durations at every slice, we simulated an application playback buffer offline for each slice at

every node to calculate the duration when there is no playback. This parameter is called the playback disruption duration. Figure 4.8(a) shows the cdf of the total playback disruption duration at every slice of all nodes for various buffer sizes. With no buffer at all (which corresponds to the 0 second buffer size), we find that the 90^{th} percentile value is 20 seconds and this value decreases steadily as the buffer size is increased. For example, with a 5 second buffer size 85% of the slices are not disrupted at all and the 90^{th} percentile disruption duration is 1 second. From this graph, we infer that most of the disruptions are of short duration and can be recovered using a buffer of modest sizes. The heavy tail in the graph was due to one particular slice of a node for which it was not able to find a parent as the bloom filter condition yielded false positives for the parents which could have supplied the slice. An obvious solution to prevent this from happening is to either request for more neighbors or join all over again.

To better show this fact, we observe the cdfs of the percentage of disruption duration over the lifetime of nodes in the system, for various levels of redundancy in Figures 4.8(b) and 4.9(a). For example, at a redundancy of 1 slice, a node is said to be disrupted if its playback buffers are disrupted for at least two slices. With no buffer at all, almost 60% of the nodes are disrupted at the first slice for more than 60% of the time. But as more redundant slices are added, we find that the disruption percentage decreases. In particular, with a redundancy of 4 slices, 90% of the nodes are not disconnected at all. Further, with a 5-second buffer, we find that no node (barring the heavy tail) is disrupted for more than 10% of its lifetime, as can be observed in Figure 4.9(a). From these graphs, we observe the tradeoff between the buffer size, redundancy and the playback disruption duration, which is fundamental to any streaming protocol.

Emulation: We have also made small deployment experiments in Emulab and have tested our protocol on a cluster of machines. The system was tested on 200 nodes emulated on a set of 50 machines, with the delays obtained from a 100-router transit-stub graph. A 100 Kbps stream was split into eight 12.5 Kbps streams and sent across multiple trees. The stream was multicast by the true source after it received its first set of 8 neighbors. As a first step, we have used hop length as the latency reduction parameter[5]. The system was run for 20 minutes and a snapshot of the data was taken at the 10^{th} minute. we chose a moderate level of heterogeneity with the degree distributed uniformly between 8 and 40 neighbors. Figures 4.9(b) and 4.9(c) show the load distributions and hop lengths for the *Lat0*, *Lat1* and the *Lat2* cases. The trends in the graphs are quite similar to the ones that we had obtained in our simulations.

Intuition on tit-for-tat: Till now, we have assumed that nodes do not lie about their loads to each other. In some environments, however, there may be free-loaders. Chunkyspread provides a natural framework for applying incentive-based constraints. To build an intuition as to how tit-for-tat may affect load and latency, we simulated a simple "weak" tit-for-tat model whereby the volume received from each neighbor must be at least within some percentage of the volume sent to that neighbor [6]. For instance, with 25% tolerance, a node that supplies 4 slices to its neighbor requires that it serves at least 3 slices back. 3-2 or 2-1 ratios are not allowed. In addition, nodes assign an initial small *credit* to new neighbors, to allow the parent-child relationships to get started, and give additional credits over time if a neighbor sends more than is received. Tit-for-tat constraints are enforced only when the credits are used up.

[5]Swaplinks does not retrieve locality-aware neighbors, hence hop length can still be a reasonable parameter.

[6][77] and [84] argue that strict tit-for-tat is impractical, and our simulations corroborate this.

We tested how this simple tit-for-tat scheme works with the Lat2 parameters. We used a 10000-node static, homogeneous setting in which each node has a target load of 16 slices. Each node periodically checks whether any of its neighbors is violating tit-for-tat, and withdraws uploaded slices as necessary. Only parent switches that fall within tit-for-tat constraints are allowed. We compared tit-for-tat ratios of 50%, 33% and 25% (corresponding to 1:2, 2:3 and 3:4 relationships respectively). We find that decreasing the ratio improves load balance, but at the expense of latency. For instance, the 90th percentile average overlay stretch for 50% tit-for-tat is 2, while for 33% it is 2.7. There are also longer and more frequent disconnections in the 25% case than in the 50% case. These experiments are encouraging in that they show that tit-for-tat constraints can be incorporated to an extent, though at the expense of other performance measures.

CHAPTER 5

CONCLUSION

In this thesis, we presented two protocols that work well with applications that require high bandwidth capacities though both of them are more generic and can work with low-volume applications as well.

PLT is based on the premise that long-haul organizational networks run either over leased fiber or VPNs, and can support DiffServ features available in current routers. We have shown that PLT, with the help of priority queuing in routers can show more protocol aggressiveness without affecting existing fairness guarantees. PLT, FAST, and XCP all perform well at near-zero loss rates. As loss rates increase, however, FAST, and to a lesser degree XCP, degrade sooner than PLT. PLT mice complete faster than FAST or XCP, as long as HCM traffic does not congest the bottleneck. PLT is fair to TCP flows in that it allows TCP flows to operate as they would in the absence of PLT. Broadly stated, PLT clearly outperforms FAST across a wide range of scenarios. While the performances of PLT and XCP are more similar, PLT is easier to deploy, because it requires no changes to network infrastructure. We have also shown through Emulab experiments on our user-space implementation that a PEP deployment is viable. More effort has to be directed towards a wide-area pdeployment of the protocol. It will also be interesting to see how PLT performs on various kinds of applications.

Chunkyspread represents a new point in the P2P multicast design space: one that has the efficiencies associated with tree-based multicast and the scalability and much of the simplicity associated with swarming-style multicast. At the foundation of Chunkyspread is the ability to build random sparse over-

lay graphs with tight statistical control over heterogeneous node degrees. This foundation, combined with a simple loop-detection mechanism based on bloom filters, provides a framework whereby different constraints and optimizations can be emphasized, depending on the application.

To date, we have focused on large-scale, non-interactive applications like the broadcast of a sporting event, at a range of volumes (text, audio, or video formats). Here, control over load is more important than latency, though in this chapter we show nevertheless that significant improvements in latency can be made if load control is relaxed slightly. We also show apples-to-apples comparisons with Splitstream, and find that Chunkyspread performs better across the board, and significantly better with respect to control over load. More experiments should be conducted in the wide area to analyze its performance. There have been works after Chunkyspread that have compared swarming protocols with Chunkyspread ([92], [90]) and have observed the tradeoffs that we had qualitatively indicated in this thesis. Being a multicast protocol, Chunkyspread should also be tested with other applications such as event notification. With more tests, both Chunkyspread and PLT can see a wide area deployment and can benefit applications using them.

Recent interest in cloud computing applications is redefining the communication model over the Internet just like p2p systems did a few years back or the client-server model in the early nineties. These applications need to give a seamless experience to the user as if she were running these applications and storing data from her machine. PLT could serve well as user connections to the cloud and as middlebox (PEP) connections within the cloud depending on where the bottleneck would lie. Of course, like in any other PEP, the middlebox

connection will help only if it runs over a network that contains the bottleneck. Chunkyspread can serve as an overlay multicast protocol in this model with the infrastructure and the users running in tandem. The load and latency control algorithms make sure that the well-provisioned nodes are usually approached for streams.

BIBLIOGRAPHY

[1] Bandwidth (computing). http://en.wikipedia.org/wiki/Bandwidth_(computing)

[2] Internet2. http://en.wikipedia.org/wiki/Internet2

[3] Google wants 'dark fiber'. http://www.news.com/Google-wants-dark-fiber/2100-1034_3-5537392.html

[4] https://network.teragrid.org/tgperf

[5] Verizon Launches Long-Haul IP VPN. http://www.xchangemag.com/hotnews/45h10124126.html

[6] GlobalCrossing. http://www.globalcrossing.com/enterprise/managed_optical/managed_optical_landing.aspx

[7] AboveNet. http://www.above.net/

[8] N. Dukkipati, N. McKeown and A. G. Fraser: RCP-AC: Congestion Control to Make Flows Complete Quickly in Any Environment. In *INFOCOM 2006*.

[9] T. J. Hacker, B. D. Noble, and B. D. Athey. The Effects of Systemic Packet loss on Aggregate TCP Flows. In *Supercomputing, ACM/IEEE 2002 Conference*.

[10] A. Snoeren, H. Balakrishnan, and F. Kaashoek. Reconsidering internet mobility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[11] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair allocations in high speed networks. In *Proceedings of SIGCOMM*, September 1998.

[12] S. Ratnasamy, S. Shenker, and S. McCanne. Towards an Evolvable Internet Architecture. In *Proceedings of SIGCOMM*, August 2005.

[13] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of SIGCOMM*, August 2002.

[14] *When 99% Isnt Quite Enough: Case Study*. e2epi.internet2.edu/case-studies/EDUCAUSE/EDUCAUSEcs.pdf

[15] Internet Study 2007: http://www.ipoque.com /media/internet_studies/internet_study_2007

[16] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM 1988*.

[17] V. Jacobson. Modied TCP Congestion Avoidance Algorithm. Technical report, 1990 (ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt).

[18] J. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes, 1995. Master's thesis, MIT.

[19] S. Floyd. SACK TCP: The sender's congestion control algorithms for the implementation sack1 in LBNL's ns simulator (viewgraphs). Technical report, Mar. 1996. Presentation to the TCP Large Windows Working Group of the IETF, 1996. (ftp://ftp.ee.lbl.gov/talks/sacks.ps).

[20] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. In SIGCOMM Computational Communications Review, Vol. 26, No. 3. (July 1996), pp. 5-21.

[21] K. Tan, J. Song, Q. Zhang and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. Technical Report, MSR-TR-2005-86, Microsoft Research.

[22] AT&T : Enterprise Business: www.business.att.com.

[23] FastSoft: www.fastsoft.com.

[24] Venturi wireless: www.venturiwireless.com.

[25] Packeteer: www.packeteer.com.

[26] S. Blake, D. Black, M. Carlson, E. Davie, Z. Wang, and W. Weiss. An Architecture for Differentiated Services, RFC 2475, December 1998.

[27] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A user-level tcp stack. http://nms.lcs.mit.edu/kandula/data/daytona.pdf, 2002.

[28] National Lambda Rail, http://www.nlr.net.

[29] Abilene Backbone Network, http://abilene.internet2.edu.

[30] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *The Proceedings of ACM SIGCOMM*, 2002.

[31] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, architecture, algorithms, performance. To appear in *IEEE/ACM Trans. On Networking*, 2007.

[32] T. V. Lakshman and U. Madhow. The performance of TCP/IP for neworks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. on Networking, 5(3):336-350*, June 1997.

[33] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Theoughput: A Simple Model and its Empirical Validation. In *The Proceedings of SIGCOMM*, 1998.

[34] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *The Proceedings of SIGCOMM* 1994.

[35] S. Floyd. Highspeed TCP for large congestion windows. RFC 3649, December 2003.

[36] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. In *The Proceedings of ACM SIGCOMM*, 2003.

[37] C. Jin, D. X. Wei, and S. H. Low. The case for delay-based congestion control. In *The Proceedings of IEEE Computer Communication Workshop (CCW)*, October 2003.

[38] I. F. Akyildiz, G. Morabito, and S. Palazzo. TCP-Peach: A New Flow Control Scheme For Satellite Networks. In *IEEE/ACM Trans. on Networking (TON)*, 2001.

[39] V. Padmanabhan and R. Katz. TCP Fast Start: A technique for speeding up web transfers. In *The Proceedings of GLOBECOMM 1998 Internet Mini-Conference*.

[40] J. Waldby, U. Madhow, and T. V. Lakshman. Total Acknowledgements: A Robust Feedback Mechanism for End-to-end Congestion Control. (Extended Abstract) *In SIGMETRICS 1998: 274:275*.

[41] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant, In *PFLDnet 2005*.

[42] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks, In *The Proceedings of IEEE INFOCOM* 2004.

[43] D. Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks, In *The Proceedings of PFLDnet* 2004.

[44] R. Wang, K. Yamada, M. Y. Sanadidi, and M. Gerla. TCP with sender-side intelligence to handle dynamic, large, leaky pipes. *IEEE Journal on Selected Areas in Communications, 23(2):235-248*, 2005.

[45] S. Bhandarkar, S. Jain, A. L. N. Reddy, and Kohler. Improving TCP Performance in High Bandwidth High RTT Links Using Layered Congestion Control. In *The Proceedings of PFLDNet* 2005.

[46] R. King, R. Riedi, and R. Baraniuk. Evaluating and Improving TCP-Africa: An Adaptive and Fair Rapid Increase Rule for Scalable TCP. In *PFLDnet* 2005.

[47] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *The Proceedings of OSDI*, 2002.

[48] A. Kuzmanovic and E. W. Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. In *The Proceedings of IEEE INFOCOM* 2003.

[49] A. Kuzmanovic, E. W. Knightly, and R. Les Cottrell. HSTCP-LP: A Protocol for Low-Priority Bulk Data Transfer in High-Speed High-RTT Networks. In *Proceedings of PFLDnet* 2004.

[50] G. Appenzeller, I. Keslassy and N. McKeown. Sizing Router Buffers. In *The Proceedings of ACM SIGCOMM*.

[51] RFC 793. Transmission Control Protocol. DARPA Internet Program Protocol Specification, September 1981.

[52] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475. An Architecture for Differentiated Service.

[53] S. E. Deering. Multicast Routing in a Datagram Internetwork. PhD thesis, Stanford University, December 1991.

[54] H. Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8):5460, 1994.

[55] J. Gemmell, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. RFC 3208: Pragmatic General Multicast.

[56] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, IEEE/ACM Transactions on Networking, December 1997.

[57] M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Low-Latency Multicast for Scalable Time-Critical Services. In *The Proceedings of the Fourth Usenix Symposium on Networked Systems Design and Implementation, Cambridge, MA, 2007.* .

[58] Conviva, www.conviva.com.

[59] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *The Proceedings of ACM NOSSDAV*, Miami Beach, FL, USA May 2002.

[60] www.veoh.com.

[61] X. Zhang, J. Liu, B. Li and T.-S. P. Yum. DONet/CoolStreaming: A data-driven overlay network for live media streaming, in *The Proceedings of INFOCOM*, Miami, FL, USA, March 2005.

[62] V. Venkataraman, K. Yoshida, and P. Francis. Chunkyspread: Heterogeneous Unstructured End System Multicast. In *The Fourteenth IEEE International Conference on Network Protocols (ICNP)*, Nov 2006.

[63] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *The Proceedings of ACM SOSP 2001*, Oct. 2001.

[64] www.napster.com.

[65] www.akamai.com.

[66] V. Vishnumurthy and P. Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. In *Proceedings of IEEE Infocom*, Barcelona 2006.

[67] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, November 2001.

[68] P. Francis. Yoid: Extending the Internet Multicast Architecture. *http://www.icir.org/yoid/*.

[69] Y. Chu, S.G. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[70] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data DisseminationUsing an Overlay Mesh In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP'03)*.

[71] M. Castro, P. Druschel, A. M. Kermarrec, and A. Rowstron, SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in Commmunications (JSAC)*, 2002.

[72] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP*, 2003.

[73] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *The Fourth International Workshop on Peer-to-Peer Systems*, February 2005.

[74] B. Cohen. Incentives Build Robustness in BitTorrent. In *The First Workshop on Economics of Peer-to-peer Systems*, June 2003.

[75] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom' 96, San Francisco, CA*.

[76] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points. In *The Proceedings of ACM SIGCOMM*, August 2004.

[77] Y. H. Chu, J. Chuang, and H. Zhang. A Case for Taxation in Peer-to-Peer Streaming Broadcast. In *ACM SIGCOMM Workshop on Practice and Theory of Incentives and Game Theory in Networked Systems (PINS)*, August 2004.

[78] A. R. Bharambe, S. G. Rao, V. N. Padmanabhan, S. Seshan, and H. Zhang. The Impact of Heterogeneous Bandwidth Constraints on DHT-Based Mul-

ticast Protocols. In *The Fourth International Workshop on Peer-to-Peer Systems,* February 2005.

[79] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. In *USENIX USITS,* 2003.

[80] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *The Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, 2002.

[81] T. W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-Compatible Peer-to-Peer Multicast. In *The Second Workshop on the Economics of Peer-to-Peer Systems,* July 2004.

[82] A. Whitaker and D. Wetherall. Forwarding without loops in Icarus. In *Proceedings IEEE OPENARCH*, 2002.

[83] W. A. Montgomery. Techniques for packet voice synchronization. In *IEEE J Select Areas Commun* 6(1):10221028.

[84] K. Tamilmani, V. Pai, and A. E. Mohr. SWIFT: A system with incentives for trading. In *Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[85] P. A. Chou, H. J. Wang, and V. N. Padmanabhan. Layered Multiple Description Coding. In *IEEE Packet Video Workshop*, Nantes, France, April 2003.

[86] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. OToole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Usenix Sympp on Operating System Design and Implementation (OSDI 2000)*, October 2000.

[87] D. A. Helder, and S. Jamin. End-host multicast communication using switch-tree protocols. In *Proceedings of the 2nd Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems (GP2PC)* May 2002.

[88] P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proc. of the Intl. Conf. on Dependable Sys. and Networks (DSN 2001)*, July 2001.

[89] Y. Tang, M. Zhang, J. Luo, Y. Zhong. Experience on Peer-to-Peer based Live

Video Streaming. In *Proceedings of 12th International Multimedia Modelling Conference,* pp.80-87, Beijing, China, January 2006.

[90] N. Magharei, R. Rejaie, and Y. Guo. Mesh or multiple-tree: A comparative study of p2p live streaming services. In IEEE INFOCOM,2007

[91] A. Nandi, B. Bhattacharjee, and P. Druschel. Understanding the design tradeoffs for cooperative streaming multicast. In *Technical report MPI-SWS-2009-002, Max Planck Institute for Software Systems*, April 2009.

[92] F.Wang, Y. Xiong, and J.Liu. mTreebone: A hybrid tree/mesh overlay for application-layer live video multicast. In ICDCS 2007.

[93] Private conversation with Bruce Davie.